C++

FAQ Lite

Table of Contents

C++ FAQ Lite

[1] 复制许可

FAQs in section [1]:

1.1 作者

Marshall Cline cline@parashift.com

（简体中文版翻译：申旻

nicrosoft@sunistudio.com）

1.2 版权布告

原文：

The entire *C++ FAQ Lite* document is Copyright © 1991-2000 Marshall P. Cline, Ph.D..

译文：

The entire *C++ FAQ Lite* document is Copyright © 1991-2000 Marshall P. Cline, Ph.D.允许复制。

译注：上述译文，仅供参考，一切请以原文为准。译者对它们亦概不负责。

1.3 复制许可

原文：

If all you want to do is quote a small portion of *C++ FAQ Lite* (such as one or two FAQs) in a larger document, simply attribute the quoted portion with something vaguely similar to, "From Marshall Cline's *C++ FAQ Lite* document, `www.parashift.com/c++-faq-lite/` ".

If you want to make a copy of large portions and/or the entire *C++ FAQ Lite* document for your own personal use, you may do so without restriction (provided, of course, that you don't redistribute the document to others, or allow others to copy the document).

If you want to redistribute large portions and/or the entire *C++ FAQ Lite* document to others, whether or not for commercial use, you must get permission from the author first (and that permission is normally granted; note however that it's often easier for you to simply tell your recipients about the one-click download option). In any event, all copies you make must retain verbatim and display conspicuously all the following: all copyright notices, the Author section, the Copyright Notice section, the No Warranty section, the *C++-FAQ-Lite* != *C++-FAQ-Book* section, and the Copy Permissions section.

If you want more and/or different privileges than are outlined here, please contact me, cline@parashift.com. I'm a very reasonable man...

译文：

如果你只是在一个大文档中引用*C++ FAQ Lite*的一小部分（如一个或两个FAQ），那么只需要类似这样表明即可："From Marshall Cline's *C++ FAQ Lite* document, `www.parashift.com/c++-faq-lite/` "。

如果你想将大部分和/或整个*C++ FAQ Lite*文档为你自己所用，那么没有限制（当然，你不能再分发此文档给别人，不允许其他人拷贝此文档）。

如果你要在分发大部分和/或C++ FAQ Lite文档给别人，那么无论是否用于商业用途，你必须首先得到作者的准许（一般情况下，授权会被许可；然而注意，对你来说可能简单地告诉收件人单击下载更容易）。在任何情况下，你必须保持完整并且显著地显示如下所有：所有版权声明，作者部分，版权布告部分，免责事项部分，*C++-FAQ-Lite* != *C++-FAQ-Book*部分和复制许可部分。

如果你想要比以上所列的更多的和/或特殊的权利，请联系我，cline@parashift.com。我是通情达理之人……

译注：上述译文，仅供参考，一切请以原文为准。译者对它们亦概不负责。

1.4 免责事项

THIS WORK IS PROVIDED ON AN "AS IS" BASIS. THE AUTHOR PROVIDES NO WARRANTY WHATSOEVER, EITHER EXPRESS OR IMPLIED, REGARDING THE WORK, INCLUDING WARRANTIES WITH RESPECT TO ITS MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

1.5 商标

- Java is a trademark of Sun Microsystems, Inc. in the United States and other countries.
- All other trademarks are the property of their respective owners.

1.6 *C++-FAQ-Lite* != *C++-FAQ-Book*

这份 *C++ FAQ Lite* 文档和 *C++ FAQ Book* 不一样。该书（C++ FAQ，第二版，Cline，Lomow，and Girou，Addison-Wesley）比这份文档大 500%，并且书店有售。细节请察看 [3]。

[2] 在线站点分发本文档

FAQs in section [2]:

- [2.1] 访问这份文档最近的镜像在那儿？
- [2.2] 如何得到 *C++ FAQ Lite* 的所有HTML文档的拷贝以便我离线阅读？
- [2.3] 如何得到 *C++ FAQ Lite* 的所有纯文本文档的拷贝以便我离线阅读？
- [2.4] 为什么通过email下载？而不是通过ftp？
- [2.5] 何处可以下载到该在线文档的繁体中文版？
- [2.6] 何处可以下载到该在线文档的葡萄牙语版？
- [2.7] 何处可以下载到该在线文档的法语版？
- [2.8] 何处可以下载到该在线文档的俄语版？

2.1 访问这份文档最近的镜像在哪儿？

选择近的站点可能提高你的访问速度：

- 美国： `www.parashift.com/c++-faq-lite/`
- 美国 #2： `www.awtechnologies.com/bytes/c++/faq-lite/`
- 加拿大： `new-brunswick.net/workshop/c++/faq`
- 芬兰： `www.utu.fi/~sisasa/oasis/cppfaq/`
- 法国： `caor.ensmp.fr/FAQ/c++-faq-lite/`
- 德国： `www.informatik.uni-konstanz.de/~kuehl/cpp/cppfaq.htm`
- 爱尔兰： `snet.wit.ie/GreenSpirit/c++-faq-lite/`
- 西班牙： `geneura.ugr.es/~jmerelo/c++-faq/`
- 台湾： `www.cis.nctu.edu.tw/c++/C++FAQ-English/`
- 中国大陆： `www.sunistudio.com/cppfaq/` `（简体中文版，译者加）`

2.2 如何得到 *C++ FAQ Lite* 的所有HTML文档的拷贝以便我离线阅读？

这里是你如何通过Email获得的打包压缩过的 *C++ FAQ Lite* HTML文件的拷贝的方法：

1. 选择一种格式（ `.zip` 通用于 Windows 和 Mac， `.tar.Z` 和 `.tar.gz` 通用于 UNIX），然后点击下面的相关按钮（只点击一次）。你不会在浏览器中看到确认页（尽管有些浏览器会显示一个 e-mail 窗口，如果是这样的话，点击“发送”）：

如果这样不行的话，则发一封 e-mail 到 `cline-cpp-faq-html-zip@crynwr.com` `(.zip)`，`cline-cpp-faq-html-tarz@crynwr.com` `(.tar.Z)` 或者 `cline-cpp-faq-html-targz@crynwr.com` `(.tar.gz)` [email的内容和主题无关紧要]

2. 等待几分钟，然后检查你的e-mail。如果没有收到包含打包的FAQ的e-mail，就多等一会儿再检查。如果等了整整一天仍然没有收到，你可以给我发e-mail或再试一次。

3. 一旦你收到了e-mail中的FAQ，按e-mail所含消息的指示接压缩FAQ。

约束：你必须仍然遵守版权布告和复制许可。尤其是，未获作者许可时，不得重分发*C++ FAQ Lite*给其他人。如果你要重分发C++ FAQ Lite给其他人，最简单的方法是告诉他们这个点击下载的特征，让他们获得自己的拷贝。

约束：FAQ使用了“长文件名”。如果你的机器无法处理长文件名（例如，DOS 和/或Windows 3.x），你不能解压缩这个FAQ。而UNIX，Windows NT，Windows 95，Windows 98，和Mac都能正确处理长文件名。

注意：选择e-mail 胜于 FTP 或 HTTP。

2.3 如何得到 *C++ FAQ Lite* 的所有纯文本文档的拷贝以便我离线阅读？

纯文本版本的*C++ FAQ Lite*每月被张贴于 `comp.lang.c++` 。这些简单的文本文件是通过机械地去除 `www.parashift.com/c++-faq-lite/` 上的HTML文件的HTML标记而产生的。因此这些纯文本文件不好看，而且不能通过超链接参考，但它们基本上包含了和HTML文件相同的信息。

这里是你如何通过Email获得的打包压缩过的 *C++ FAQ Lite* HTML文件的拷贝的方法：

1. 选择一种格式（ `.zip` 通用于 Windows 和 Mac， `.tar.Z` 和 `.tar.gz` 通用于 UNIX），然后点击下面的相关按钮（只点击一次）。你不会在浏览器中看到确认页（尽管有些浏览器会显示一个 e-mail 窗口，如果是这样的话，点击“发送”）：

   如果这样不行的话，则发一封 e-mail 到 `cline-cpp-faq-plaintext-zip@crynwr.com` (.zip), `cline-cpp-faq-plaintext-tarz@crynwr.com` (.tar.Z), or `cline-cpp-faq-plaintext-targz@crynwr.com` (.tar.gz) [email的内容和主题无关紧要]

2. 等待几分钟，然后检查你的e-mail。如果没有收到包含打包的FAQ的e-mail，就多等一会儿再检查。如果等了整整一天仍然没有收到，你可以给我发e-mail或再试一次。

3. 一旦你收到了e-mail中的FAQ，按e-mail所含消息的指示接压缩FAQ。

约束：你必须仍然遵守版权布告和复制许可。尤其是，未获作者许可时，不得重分发*C++ FAQ Lite*给其他人。如果你要重分发C++ FAQ Lite给其他人，最简单的方法是告诉他们这个点击下载的特征，让他们获得自己的拷贝。

注意：选择e-mail 胜于 FTP 或 HTTP。

2.4 为什么通过email下载？而不是通过ftp？

使用FTP或HTTP会有“缓存相关（cache coherency）”的问题。

多年来，我注意到有许多过期的（不，是远古的）*C++ FAQ Lite*的拷贝流传在Internet上。由于这些远古的版本一般包含bug，遗漏的特征和过期的信息，这导致了许多的混淆。更带来了我的收件箱里的大量的无价值的email。看来我可能永远度过这个困难时期了：无论我把当前的版本做的如何有清晰，还是有成百（可能上千）的人不知道他们正在阅读一个过期的版本。这使得他们和我都很辛苦。

通过email而不是ftp下载*C++ FAQ Lite*，我能够为人们提供额外的服务：邮件机器人（发送FAQ拷贝给所有人的一个Perl脚本程序）记住了每个人所得到的版本，并且当某人的版本过期时，机器人会发送一封礼貌的邮件给他，像这样：“您的FAQ的拷贝已经过期；如果您想要升级，按这里”。（注意：我还没有建立这个功能；请耐心等待）。

这样做的目的是帮助你保持最新版本，因此你不会阅读到过期的信息。也可以使我的收件箱从充斥着阅读过期版本的困惑的读者的问题中解脱出来。

因此，请千万不要发e-mail询问FTP地址，没有。谢谢。

2.5 何处可以下载到该在线文档的繁体中文版？

`www.cis.nctu.edu.tw/chinese/doc/research/c++/C++FAQ-Chinese/` 包含了“Big5”编码的繁体中文版。注意：“Big5”是台湾使用的16位中文编码。

2.6 何处可以下载到该在线文档的葡萄牙语版？

`www.mathematica.com.br/CPPFL00.htm` 包含了FAQ的葡萄牙语的翻译。

2.7 何处可以下载到该在线文档的法语版？

`www.ifrance.com/jlecomte/c++/c++-faq-lite/` 包含了FAQ的法语的翻译。

2.8 何处可以下载到该在线文档的俄语版？

`quirks.chat.ru/cpp/faq/` 包含了FAQ的俄语的翻译。

[3] *C++-FAQ-Book* 与 *C++-FAQ-Lite*

FAQs in section [3]:

3.1 除了 *C++ FAQ Lite* 外，有*C++ FAQ Book*吗？

有，这本书是：*C++ FAQs* by Cline, Lomow, and Girou, Addison-Wesley, 1999, ISBN 0-201-30983-1。

该书获得了Amazon.com的五星评价，并且书店有售。

这是封面。

这里有一些摘录。

这里有不同的书店报价（按字母顺序排列）：

- On 7/3/00, 价格是 $33.56 at this link within AllDirect.com.
- On 7/3/00, 价格是 $39.95 at this link within Amazon.com.
- On 7/3/00, 价格是 $39.95 at this link within BarnesAndNoble.com.
- On 7/3/00, 价格是 $31.50 at this link within BookPool.com.

3.2 *C++ FAQ Lite*与*C++ FAQ Book?*有很大不同吗？

是的，非常大的不同。

数量上来说，*C++ FAQ Book* 5倍于*C++ FAQ Lite*。

最起码：书和这个C++ FAQ Lite文档不一样。无论从宽度和深度来说，书都更胜一筹——它涵盖了这个 Lite文档所不涉及的内容，并且包括了更多的细节。

除此之外，书中带有许多的程序实例——比本 Lite文档中的多得多。

[6] 综述

FAQs in section [6]:

6.1 C++ 是一种实用的语言吗？

是的。

C++ 是一种实用的工具。它不完美，但是有用。

在软件产业的世界里，C++被看作一种可靠的，成熟的，主流的工具。它得到普遍的工业支持，因而从一种全面的商业角度来看，它是“优秀”的。

6.2 C++ 是一种完美的语言吗？

不是。

C++的原设计目标不是作为完美的面向对象语言的典范。它被设计为一种实用的工具，来解决现实世界的问题。像所有的实用工具一样，它有瑕疵。不过，非完美就无用的，那是纯理论的框架。而不是C++的目标。

6.3 面向对象（OO）有什么用?

面向对象技术是我们所知道的开发大型的，复杂的软件应用和系统的最佳方法。

OO：应付大型的，复杂的软件系统时，软件工业是"失败的"。但是这种"失败"实际上归因于我们的成功：我们的成功使得用户想得到更多。不幸的是我们创造了市场的饥渴，而"结构化"的分析、设计和编程技术无法满足这种饥渴。因此需要我们创造一种更好的典范。

C++支持面向对象（OO）编程。C++ 也能够被当作传统的编程语言使用（作为"一种更好的C"）或使用。基本上每种方法都有其优点和缺点。也不要在使用一种方法时期望得到另外一种技术的好处。（最常见的误解是，如果把C++"作为一种更好的C"来使用它，那么就不要期望得到面向对象所带来的好处。)

6.4 泛型(generic)编程有什么用？

C++支持泛型编程。泛型编程是一种能够最大化代码复用而又不损失效率的一种开发软件的方法。（这里的"效率"严格来说并非必须，但有了更好。）

泛型组件非常易用，并且一般会隐藏很多复杂性，当然前提是至少要设计得好。另外一个有趣的特性是它们能够使代码运行更快，尤其是当这些组件被更多地 使用时。这是一种很好的情形：当你用这些组件来完成一些繁杂的工作时，你的代码会变得更少更简单，出错的几率也小些，同时代码执行起来还更快。

大多数开发者并没有足够的能力来开发出这些泛型组件，但却能够使用它们。开发这些组件的过程是复杂痛苦的。你不断尝试、挠头、在凌晨3点有了灵感然 后起床，不断地重写代码（不断重写，不断重写）。一句话，要不断迭代。像谚语所说，这是在往5磅容量的口袋里塞10磅东西。不喜欢思考、不喜欢解决难题的 人就不必费这劲了。

幸运的是，泛型组件是，呃，很通用的。所以你所在的单位通常不必开发很多泛型组件。有很多已经做好的泛型组件，例如STL。Boost里面有更多的组件。还有很多其它的库。

6.5 C++ 比 Ada 更好吗？（或 Visual Basic, C, FORTRAN, Pascal, Smalltalk, 或其他语言？）

停！这样的问题没有意义。在对这个问题发表不同意见之前请先阅读下文。

99%的情况下，编程语言的选择是出于商业上的考虑， 而不是技术上 的考虑。真正重要的是类似以下方面的商业上的考虑：开发机器的编程环境，目标机器是否包含运行时环境，运行时和/或开发环境的许可/法律问题，是否有受过 训练的开发者，是否有咨询服务，和企业文化/政策。它们扮演的角色一般比编译期性能，运行时性能，静态还是动态类型，静态还是绑定等更为重要。

从纯粹的技术角度争论一种语言比另一种更好的人（他们忽略了商业问题其实更重要），正是暴露了他们自己技术上的缺乏，别听他们的。商业问题比技术问题更重要，任何没有意识到这一点的人注定会做出带来糟糕后果的决策。这些人对雇主来说是危险的。

6.6 谁在用 C++?

很多很多公司和政府部门，非常多。

有大量的开发者（并且因此有大量的底层有效支持，包括厂商，工具开发者，培训等等）是C++的特征之一

6.7 学习 OO/C++ 需要多长时间？

一些公司成功地讲授标准的工业界的“短期课程”，将大学一学期的课程压缩到了一个星期40个小时。但是不论你在何处获得培训，要确保课程具有动手项 目，大多数人是在接手项目之后才将概念“凝结成形”，得以学成。即使得到最好的培训，人们也还没能准备好。（译注：指做实际的项目）

精通OO/C++需要6-12个月。如果身边有专家的话，会少些。如果没有一个“好的”通用型的C++类库，则会多一些。成为可以指导别人的专家则需要3年。

有 些人永远不行，除非你是可教的“儒子”并且有个人驱动力。可教的最低要求是，当你错了的时候必须能够承认。驱动力的最低要求是，你必须愿意投入一些额外的 时间。记住，学习一些新的东西比改变你的典范（paradigm）（即改变你思考的方法，改变你对于什么是好的认识，改变你在技术世界中的思维模式）要容易的多。

你应该做两件事：

- 找一个“指导人”
- 看两类书：一类是有关C++中什么是合法的, 另一类是有关 C++中什么是该做的。

你不应该做两件事：

- 不应该去学习 C 作为学习 OO/C++ 的台阶
- 不应该去学习 Smalltalk 作为学习 OO/C++ 的台阶

6.8 从商业角度看 C++ 有哪些特征？

从商业角度看 OO/C++ 有这样一些特征：

- C++ 有巨大的安装基础，这意味着你会有很多厂商在工具，环境，咨询服务等上提供支持，而且你可以在你的履历上加上非常有价值的一条。
- C++ 让开发者为软件块提供简化的接口，以改善这些软件块被使用（重用）时的错误率。
- C++ 通过算符重载让你利用开发者的直觉，降低重用用户的学习曲线。
- C++ 将对软件块的访问局部化，降低更改时的成本。
- C++ 减少安全性和可用性的权衡，改善使用（重用）软件块时的成本。
- C++ 减少安全性和速度的权衡，改善错误率而不丧失性能。
- C++ 给你继承和动态绑定，以便旧的代码调用新的代码，使得针对市场的快速扩展／调整你的软件成为可能。

6.9 虚函数（动态绑定）对于 OO/C++来说是主要的吗？

是的！

没有虚函数包含了许多模板以实现同样非常好的“泛型编程（译注：也称通用编程，"generic programming"）”技术，但虚函数仍然是用C++进行面向对象编程的核心。

从商业角度）。 技术人员通常认为在C和非面向对象的C++之间有很大的区别，但如果没有面向对象，这个区别通常不足以证明培训开发者，新工具等的成本是值得的。换句话 说，如果我被某个经理征询意见，是否从C转向非面向对象的C++（也就是说，转换语言而不转换典范），那么我可能会劝阻他这样做，除非有逼不得已的面向工 具的原因。从商业角度看，面向对象能使系统具有可扩展性和可适应性，但只有C++类的语法而没有面向对象的话，就不会减少维护成本，而实际上会增加培训成 本。

底线：没有虚函数的C++不是面向对象。用类编程而没有动态绑定则称为“基于对象”，而不是“面向对象”。踢出虚函数和踢出OO（译 注：即面向对象）是一回事。所剩下的就是基于对象编程了，和最初的Ada语言类似（顺便说一下，新的Ada语言支持OO而不是基于对象编程了）。

注意：在泛型编程中不需要虚函数。结合其它情况，这表明你无法通过简单地数虚函数的数量来判断所使用的编程范式

6.10 我来自密苏里州。你能给我一个理由，为什么虚函数（动态绑定）造成很大的不同？

总体来说：动态绑定能通过使旧的代码调用新的代码来提高重用。

在 OO（译注：即面向对象）之前，重用是通过使新的代码调用旧的代码来完成的。举例来说，程序员可以写一些代码来调用一些重用的代码，如 `printf()`。

在OO中，重用能够通过使旧的代码调用新的代码来完成。例如，程序员可以写一些代码被非常非常始祖的 框架所调用。而不需要修改始祖的代码。事实上，甚至不需要被重新编译。即使源代码已经遗失了25年，你只有目标文件，那个原始的目标文件将会调用新的扩展 的代码而不会遗失什么。

这是可扩展性，这是OO。

6.11 C++ 是否向下兼容 ANSI/ISO C？

差不多。

C++尽可能地兼容C，但不完全。在实践上，主要的区别是，C++需要原型，`f()` 声明一个不带参数的函数（在 C 中，`f()` 和 `f(...)` 是相同的）。

还有一些非常微小的差别，象在C++中 `sizeof('x')` 等于 `sizeof(char)`，而在 C 中等于 `sizeof(int)`。同样，C++在同一个结构的tag名和其他名称放在同一名字空间内，而在 C 中，需要显式的 `struct`（举例来说，`typedef struct Fred Fred;` 技巧当然也可以工作，但在C++中是多余的）。

6.12 C++ 标准化了吗？

是的。

C++标准被ISO（国际标准化组织）和一些国家标准组织，如ANSI（美国国家标准协会），BSI（英国标准协会），DIN（德国国家标准组织）所定稿和采用。ISO标准在1997年11月14日经投票一致被定稿和采用。

ANSI C++委员会被称为“X3J16”。ISO C++标准小组被称为“WG21”。ANSI/ISO C++标准的主要参与者几乎包含了每个人：有来自澳大利亚，加拿大，丹麦，法国，德国，爱尔兰，日本，荷兰，新西兰，瑞典，英国和美国的代表，连同大约一 百多个公司的代表和感兴趣的个人。主要参与者包括AT&T，爱立信，Digital，Borland，惠普，IBM， Mentor Graphics，微软，Silicon Graphics，Sun Microsystems和西门子。经过大约8年的工作，标准完成了。在1997年11月14日，代表们出席了在莫里森镇的投票，标准被一致认可。

6.13 何处能得到 ANSI/ISO C++ 标准的拷贝？

准备好花钱吧——该文档不是免费的。有很多种方法获得这份文档。下面列出了一些：

- 访问ANSI，搜索“14882”（或对于C标准来说，搜索“9899”）

- 访问Tech-Street，搜索“14882”（或对于C标准来说，搜索“9899”）。
- 去任何一家书店，搜索“0470846747”或“The C++ Standard, Incorporating Technical Corrigendum No. 1.”例如，这里，这里，这里
- 致电给NCITS（信息技术标准国家委员会National Committee for Information Technology Standards，这是原来叫做“X3”的组织的新名字，其发音类似“insights”）。联系人是Monica Vega，202-626-5739 或 202-626-5738。寻求FDC 14882文档。

这里有一些相关文档。虽然是免费的，不过不是标准本身。

- 社区草案#2是免费的，但不是正式的，也过时了，还可能有错误：这里和这里。
- ISO委员会的新闻发布信息在这里。非程序员也能读懂该发布信息。

6.14我可以问哪些“面试问题”来判断面试者真的懂了？

这个问题主要是针对想做好面试C++应聘者的非技术性管理人员和人力资源人员。如果你是一名准备去应聘的C++程序员，并且正在翻看本FAQ希望能够提前知道会面试什么问题，并以此来避免真正去学习C++，那么你应该感到羞耻：还是花点时间来提高自己的技术吧，这样就不必靠“作弊”来生活了！

回到非技术性管理人员和人力资源人员上：很明显你有足够的自个来判断面试者是否适合你公司的文化。不过有很多假冒内行、装腔作势唬人的家伙，所以你 需要找一些有技术实力的人一起来判断面试者的技术能力是否满足要求。很多公司雇了看上去不错但实际很废柴的人，结果深受其害。这些人虽然知道怎么回答一些 困难的问题，但本质上是不合格的。辨别这些伪专家的唯一办法是找人和你一起，这人要能够提出考验水平的技术问题。单凭自己是不行的。即使我给你一摞“迷惑 人的问题”，还是不能辨别出那些不良分子。

你的技术伙伴可能没有（通常是没有）足够的资格来判断面试者的个性或软实力，所以在决策过程中，请不要放弃你做为最终决定者的权利。但同时也请不要认为你能够通过问一些C++问题，就能获得一些粗略的线索，知道应聘者是否真的明白他所说的东西。

已经说过了：如果你有足够技术能力来阅读这份FAQ，那么你能够在这里找到很多好的面试问题。这份FAQ包含很多好问题来区分良莠。这份FAQ主要探讨程序员应该做什么，而不只是编译器允许程序员做什么。有很多C++里可以做但不应该做的事情。这份FAQ帮助人们区分这两者。

6.15当FAQ说“这些是邪恶的”，是什么意思？

这表示这些是你在大部分时候要避免的，但不是在所有时候都要避免的。例如，最后当某些东西没有其替代方案邪恶时。你会采用这些”邪恶“的东西。好吧，这是开玩笑的。别太当真。

这个词的真正意图（我听到你说“啊哈，果然有隐情！”。你是对的，的确有）是让C++新手摆脱一些旧有的思想。例如，C程序员开 始用C++时常常会过多使用指针、数组和/或#define。本FAQ将这些东西归为“邪恶”，以便给予C++新手一个往正确方向上的有力（同时也是古怪 有趣的）推动。类似“指针很邪恶”这种搞笑说法是为了说服C++新手C++“并非除了傻傻的//注释在其它方面就很像C了”。

现在说点正经的。我并不是说宏或数组或指针是要谋杀或绑架。呃，可能指针会干这些事（开个玩笑！）。所以不要对“邪恶”这个词太过敏感：用这个词只是为了为了听上去有些夸张。所以不要试图寻找有关哪些是“邪恶”或“不邪恶”的精确技术性定义：压根就没有。

还有一件事要注意：被称作“邪恶”的东西（宏、数组、指针等）并非在所有情况下总是邪恶的。当它们“不是最坏”的选择时，就用它们。

6.16有时会用到那些“邪恶”的东西么？

当然会！

一种尺寸不可能适用所有情况。停！现在，找个尖头的笔在你的眼镜内侧写上：“软件开发就是做决策。“思考”(think)不是一个4个字母的单词。在软件中很少有“从不”和“总是”之类实施起来不需要动脑子的规则，没有那些在所有情况下都适用的规则，没有那种放之四海皆准的规则。

所以最终你将会使用那些“邪恶”的技术。如果你觉得这个词不舒服，那么换成“很多时候不推荐”（不过可别辞职改行当作家：像那种软蛋类型的术语只会让人睡着:-)。

译注：这里的“软蛋”术语指”很多时候不推荐“这种听上去不够”硬“的表达。

6.17是否知道这些东西的技术定义很重要：“好的OO”，“好的类设计”？

可能你不喜欢，但简短的回答是，“不”（注意这个回答是给实践者而不是理论家的）。

专业的软件设计者根据这些来做评估：业务需求（时间，金钱和风险）以及技术需求（例如是不是“好的OO”或“好的类设计”）。这要困难很多，因为除 了技术因素，还涉及到业务问题（期限，人员的技术能力，知道公司的发展方向以便决定如何灵活设计，是否愿意考虑将来可能的变化（指实际可能发生而不只是理 论上可能）等等。）。然而，这样作出的决策更加可能带来好的效益。

做为一名开发者，你对老板要负有一种信任上的责任，要只投资那种能够带来可观回报的方面。如果除了技术问题之外不再问业务问题，你作出的决策就可能带来不可预知的商业结果。

不管喜不喜欢，这意味着实际上你可能最好还是不要去定义诸如“好的类设计”和“好的OO”这类术语。实际上我相信这些精确的、纯技术的定义可能会非 常危险，可能会浪费公司钱财，最终甚至会把人们的工作也搭进去。听上去有些夸张，但这是有一个很好的理由的：如果这些术语以一种精确、纯技术的方式来定 义，那么开发者可能就会好心办坏事，而忽略业务上的考虑，因为他们想要达到这些纯技术定义的“良好”标准。

任何纯技术定义的“良好”，例如好的“OO”或“好的设计”或是任何其它不需考虑期限、业务目标（即投资方向）、预期的未来变化、企 业在未来的投资意愿方面的文化、做维护工作的团队的技术水平等事情的东西，都是危险的。因为这回诱使程序员相信他们做的决定是“正确”的，而实际却可能导致灾难性后果。或者也可能虽然没有带来糟糕的商业后果，但关键是：当你在做决定时忽略了业务，那么最终结果会是随机的，有点无法预期。这就不好了。

事实很简单，业务问题高于技术问题。任何有关“好”的定义，如果不承认这点，那这个定义就是糟糕的。

6.18 当人们抱怨说“FAQ”这个词太误导人了，因为它强调问题而不是答案，因此我们需要换一个不同的缩写词，这时我应该怎么对他们说？

告诉他们成长起来。

有人希望能够把”FAQ“换一个词，比如要能够强调答案而不是问题。但单词或短语是根据其用法来定义的。很多人已经理解了“FAQ”这个词本身的含义了。最好把它当作是一个名字，而不是缩写词。做为一个单词来说，“FAQ”已经表示一份常见问题和答案的列表了。

这并不是鼓励用词时不加考虑。相反，关键是清晰的交流需要使用人们已经理解的词汇。争论我们是否应该为“FAQ”换个词很蠢，而且浪费时间。如果这个词还没有为大家所熟知的话，那就是另外一回事了。但当很多人都已经理解了再去更换，那就没有意义了。

举个（不太完美）的类比，大家已经广泛接受'\n'做为换行符了，可现在仍然有少数程序员和某种计算机打交道，这些计算机有实际“换行”的打字终端。没人会在乎这个的。这就是个换行符，别在为之烦恼了。同理，'\r'是回车符，即使你的机器没有这种东西。接受它吧。

另一个（不完美）的类比是RAII。感谢Andy Koenig等人的伟大工作，“RAII”在C++社区已经广为人知了。“RAII”代表了一种非常有价值的概念，并且你应该经常用它。但是，如果你把“RAII”做为一个首字母缩写词来分解，并且如果你仔细研究组成这个缩写词的各个单词，你会意识到这些词并不能完美代表其含义。但谁在乎呢？！？重要的是概念，“RAII”只不过是其背后概念的一个称呼。

> 细节：如果你把RAII分解成各个单词（Resource Acquisition Is Initialization获取资源即初始化），你可能会认为RAII是指在初始化时获取资源。然而，RAII的威力并非来自将获取和初始化绑在一起，而使在于将回收资源和析构联 系在一起。更精确的缩写可能是“RRID”（Resource Reclamation Is Destruction资源回收即析构），或者DIRR（Destruction Is Resource Reclamation析构即回收资源），但既然大家已经广泛理解了RAII，使用这个词就比抱怨术语更重要。RAII是一种思想的名称，其做为一个缩写 词的精确性就不那么重要了。
>
> 所以还是把“FAQ”当作一个名字，其含义已被广泛接受了。单词的意义是由其用法定义的。

对应原文最后更新2009年1月2日 翻译最后更新2009年3月28日

[7] 类和对象

FAQs in section [7]:

- 7.1] 类是什么[?
- [7.2] 对象是什么?
- [7.3] 什么样的接口是“好”的?
- [7.4] 封装是什么?
- [7.5] C++是如何在安全性和可用性间取得平衡的?
- [7.6] 我如何才能防止其它程序员查看我的类的私有部分而破坏封装?
- [7.7] 封装是一种安全装置吗?
- [7.8] 关键字 struct 和 class 有什么区别?

7.1 类是什么?

面向对象软件的基本组成物

类定义数据类型,就如同 C 中的结构。从计算机科学的角度来理解,类型由状态集合和转换这些状态的操作集合组成。因为 `int` 既有状态集合,也有象 `i + j` 或 `i++` 等这样的操作,所以 `int` 是一种类型。同样,类提供了一组操作集合(通常是 `public:`)和一组描述类型实例所拥有的抽象值的数据集合。

可以将 `int` 看作为一个有 operator++ 等成员函数的类。(`int` 实际并不是一个类,但是基本类似:一个类是一种类型,就如同 `int` 是一种类型)

注意: C 程序员可以将类看作为成员默认为私有的结构。但是,如果那是你对类的全部认识,那么你可能要经历个人的思考模式的转变了。

7.2 对象是什么?

和语义有关的存储区域

当我们声明了 `int i` ,我们说:“ `i` 是 `int` 类型的一个对象”。在 OO/C++ 中,“对象”通常意味着“类的一个实例”。因此,类定义多个对象(实例)的可能的行为。

7.3 什么样的接口是“良好”的?

提供了一个将“块”状的软件简化了的视图,并且以“用户”的词汇表达的接口。(“块”通常是一个或一组紧密相连的类;“用户”是指其它的开发者而不是最终客户)

- “简化了的视图”指隐藏不必要的细节。这样可以减少用户的错误率。
- “用户的词汇”意思是用户不需要学习新的词汇或概念，这样可以降低用户的学习曲线。

7.4 封装是什么？

防止未被授权地访问一些信息和功能。

节省成本的关键是从软件“块”的稳定部分中分离出可变的部分。封装给这个“块”安置了防火墙，它可以防止其它“块”访问可变的部分；其它“块”仅仅能够访问稳定的部分。这样做，当可变的部分改变后，可以防止其它“块”被破坏。在面向对象软件的概念中，“块(chunk)”通常指一个或一组紧密相连的类。

“可变的部分”是实现的细节。如果“块”是单个类，那么可变的部分通常用 `private:` 和/或 `protected:` 关键字来封装。如果“块”是一组紧密相连的类，封装可被用来拒绝对组中全部类的访问。继承。

“稳定的部分”是接口。好的接口提供了一个以用户的词汇简化了的视图，并且被从外到里的设计。（此处的“用户”是指其它开发者，而不是购买完整应用的最终用户）。如果“块”是单个类，接口仅仅是类的 public: 成员函数和友元，那么接口可以包括模块中的多个类。

设计一个清晰的接口并且将实现和接口分离，只不过是允许用户使用接口。而封装实现可以强迫用户使用接口。

7.5 C++是如何在安全性和可用性间取得平衡的？

在 C 中，封装是通过在编辑单元或模块中，将对象声明为静态来完成的。这样做防止了其他模块访问静态区域。（顺便说一句，现在这种做法是被遗弃的：不要在 C++中这样做）

不幸的是，由于没有对一个模块的静态数据产生多个实例的直接支持，这种处理方法不支持数据的多个实例。在 C 中如果需要多个实例，那么程序员一般使用结构。但是很不幸，C 的结构不支持封装。这增加了在安全性（信息隐藏）和可用性（多实例）之间取得平衡的难度。

在 C++中，你可以利用类来同时获得多实例和封装性。类的 public: 部分包含了类的接口，它们通常由类的 public: 成员函数和它的友元 部分包含了类的实现，而通常数据就在这里。

最终的结果就象是“封装了的结构”。这样就易于在安全性（信息隐藏）和可用性（多实例）间取得平衡。

7.6 我如何才能防止其它程序员查看我的类的私有部分而破坏封装？

不必这么做——封装是对于代码而言的，而不是对人。

只要其它程序员写的代码不依赖于他们的所见，那么即使它们看了你的类的 `private:` 和/或 `proteced:` 部分，也不会破坏封装。换句话说，封装不会 阻止人认识类的内部。封装只是防止他们写出依赖类内部实现的代码。你的公司不必为维护你眼睛所看到的东西支付维护成本，但是必须为维护你的指尖写出的代码支付维护成本。正如你知道的，倘若他们写的代码依赖于接口而不是实现，就不会增加维护成本。

此外，这很少成为一个问题。我想不会有故意试图访问类的私有部分的程序员。My recommendation in such cases would be to change the programmer, not the code" [James Kanze; used with permission].

7.7 封装是一种安全装置吗？

不。

封装 != 安全。

封装要防止的是错误，而不是间谍。

7.8 关键字 struct 和 class 有什么区别？

struct 的成员默认是公有的，而类的成员默认是私有的。注意：你应该明白地声明你的类成员为公有的、私有的、或者是保护的，而不是依赖于默认属性

struct 和 class 在其他方面是功能相当的。

OK，明晰的技术谈论够多了。从感情上讲，大多数的开发者感到类和结构有很大的差别。感觉上结构仅仅象一堆缺乏封装和功能的开放的内存位，而类就象活的并且可靠的社会成员，它有智能服务，有牢固的封装屏障和一个良好定义的接口。既然大多数人都这么认为，那么只有在你的类有很少的方法并且有公有数据（这种事情在良好设计的系统中是存在的!）时，你也许应该使用 struct 关键字，否则，你应该使用 class 关键字。

[8] 引用

FAQs in section [8]:

8.1 什么是引用?

对象的别名(另一个名称)。

引用经常用于“按引用传递(pass-by-reference)”：

```
void swap(int& i, int& j)
{
  int tmp = i;
  i = j;
  j = tmp;
}

int main()
{
  int x, y;
  _// ..._
  swap(x,y);
}
```

此处的 `i` 和 `j` 分别是main中的 `x` 和 `y`。换句话说，`i` 就是 `x` —— 并非指向 x 的指针，也不是 `x` 的拷贝，而是 `x` 本身。对 `i` 的任何改变同样会影响 `x`，反之亦然。

OK，这就是作为一个程序员所认知的引用。现在，给你一个不同的 角度，这可能会让你更糊涂，那就是引用是如何实现的。典型的情况下，对象 `x` 的引用 `i` 是 `x` 的机器地址。但是，当程序员写 `i++` 时，编译器产生增加 `x` 的代码。更详细的来说，编译器用来寻找 `x` 的地址位并没有被改变。C 程序员将此认为好像是 C 风格的按指针传递，只是句法不同 (1) 将 & 从调用者移到了被调用者处，(2)消除了 `*` s。换句话说，C 程序员会将 `i` 看作为宏 `(*p)` ，而 p 就是指向 `x` 的指针（例如，编译器自动地将潜在的指针解除引用；`i++` 被改变为 `(*p)++` ；`i = 7` 被自动地转变成 `*p = 7` ）。

很重要：请不要将引用看作为指向一个对象的奇异指针，即使引用经常是用汇编语言下的地址来实现的。引用就是对象。不是指向对象的指针，也不是对象的拷贝，就是对象。

8.2 给引用赋值，意味着什么？

改变引用的“指示物”（引用所指的对象）。

请记住: 引用就是它的指示物，所以当改变引用的值时，也会改变其指示物的值。以编译器编写者的行话来说，引用是一个“左值”（它可以出现在赋值运算符左边）。

8.3 返回一个引用，意味着什么？

意味着该函数调用可以出现在赋值运算符的左边。

最初这种能力看起来有些古怪。例如，没有人会认为表达式 `f() = 7` 有意义。然而，如果 a 是一个 Array 类，大多数人会认为 `a[i] = 7` 有意义，即使 `a[i]` 实际上是一个函数调用的伪装（它调用了 如下的 Array 类的 `Array::operator[](int)`）。

```
class Array {
public:
  int size() const;
  float& operator[] (int index);
  _// ..._
};

int main()
{
  Array a;
  for (int i = 0; i < a.size(); ++i)
    a[i] = 7;    // 这行调用了 Array::operator[](int)
}
```

8.4 `object.method1().method2()` 是什么意思?

连接这些方法的调用，因此被称为方法链

第一个被执行的是 `object.method1()`。它返回对象，可能是对象的引用（如，`method1()` 可能以 `return *this` 结束），或可能是一些其他对象。我们姑且把返回的对象称为 `objectB`。然后 `objectB` 成为 `method2()` 的 `this` 对象。

方法链最常用的地方是 `iostream` 库。例如，`cout << x << y` 可以执行因为 `cout << x` 是一个返回 `cout` .的函数

虽然使用的较少，但仍然要熟练掌握的是在命名参数法（Named Parameter Idiom）中使用方法链。

8.5 如何能够使一个引用重新指向另一个对象？

不行。

你无法让引用与其指示物分离。

和指针不同，一旦引用和对象绑定，它无法再被重新指向其他对象。引用本身不是一个对象（它没有标识; 当试图获得引用的地址时，你将的到它的指示物的地址；记住：引用就是它的指示物 ）。

从某种意义上来说，引用类似 `int* const p` 这样的const指针（并非如 `const int* p` 这样的指向常量的指针）。不管有多么类似，请不要混淆引用和指针；它们完全不同。

8.6 何时该使用引用, 何时该使用指针？

尽可能使用引用，不得已时使用指针。

当你不需要“重新指向(reseating)”时，引用一般优先于指针被选用。这通常意味着引用用于类的公有接口时更有用。引用出现的典型场合是对象的表面，而指针用于对象内部。

上述的例外情况是函数的参数或返回值需要一个“临界”的引用时。这时通常最好返回/获取一个指针，并使用 NULL 指针来完成这个特殊的使命。(引用应该总是对象的别名，而不是被解除引用的 NULL 指针)。

注意：由于在调用者的代码处，无法提供清晰的的引用语义，所以传统的 C 程序员有时并不喜欢引用。然而，当有了一些 C++ 经验后，你会很快认识到这是信息隐藏的一种形式，它是有益的而不是有害的。就如同，程序员应该针对要解决的问题写代码，而不是机器本身。

8.7 什么是对象的句柄？它是指针吗？它是引用吗？它是指向指针的指针？它是什么？

句柄术语一般用来指获取另一个对象的方法——一个广义的假指针。这个术语是（故意的）含糊不清的。

含糊不清在实际中的某些情况下是有用的。例如，在早期设计时，你可能不准备用句柄来表示。你可能不确定是否将一个简单的指针或者引用或者指向指针的指针或者指向引用的指针或者整型标识符放在一个数组或者字符串（或其它键）以便能够以哈希表（hash-table）（或其他数据结构）或数据库键或者一些其它的技巧来查询。如果你只知道你会需要一些唯一标识的东西来获取对象，那么这些东西就被称为句柄。

因此，如果你的最终目标是要让代码唯一的标识/查询一个Fred类的指定的对象的话，你需要传递一个Fred句柄这些代码。句柄可以是一个能被作为众所周知的查询表中的键（key）来使用的字符串（比如，在 `std::map<std::string,Fred>` 或 `std::map<std::string,Fred*>` 中的

键），或者它可以是一个作为数组中的索引的整数（比如，`Fred* array = new Fred[maxNumFreds]`），或者它可以是一个简单的 Fred*，或者它可以是其它的一些东西。

初学者常常考虑指针，但实际上使用未初始化的指针有底层的风险。例如，如果Fred对象需要移动怎么办？当Fred对象可以被安全删除时我们如何获知？如果Fred对象需要（临时的）连续的从磁盘获得怎么办？等等。这些时候的大多数，我们增加一个间接层来管理位置。例如，句柄可以是Fred*，指向*Fred*的指针可以保证不会被移动。当Fred对象需要移动时，你只要更新指向Fred*的指针就可以了。或者让用一个整数作为句柄，然后在表或数组或其他地方查询Fred的对象（或者指向Fred对象的指针）。

重点是当我们不知道要做的事情的细节时，使用句柄。

使用句柄的另一个时机是想要将已经完成的东西含糊化的时候（有时用术语magic cookie也一样，就像这样，“软件传递一个magic cookie来唯一标识并定位适当的Fred对象”）。将已经完成的东西含糊化的原因是使得句柄的特殊细节或表示物改变时所产生的连锁反应最小化。举例来说，当将一个句柄从用来在表中查询的字符串变为在数组中查询的整数时，我们可不想更新大量的代码。

当句柄的细节或表示物改变时，维护工作更为简单（或者说阅读和书写代码更容易），因此常常将句柄封装到类中。这样的类常重载`operator->` 和 `operator*`算符（既然句柄的效果象指针，那么它可能看起来也象指针）。

[9] 内联函数

FAQs in section [9]:

9.1 内联函数有什么用？

当编译器内联展开一个函数调用时，该函数的代码会被插入到调用代码流中（概念上类似于展开 `#define` 宏）。这能够改善性能（当然还有很多其它因素），因为优化器能够顺序集成(procedurally integrate)被调用代码，即将被调用代码直接优化进调用代码中。

有几种方法将一个函数设定为内联。其中一些需要使用 `inline` 关键字)，还有一些则不需要。不管你用何种方法设定函数为内联，这只是个请求，而编译器可以忽略它。编译器可能会展开内联函数调用，也可能不展开。（这看上去非常模糊，但不要为之沮 丧。这种灵活性其实有很大优点：这可以让编译器能够区别对待很长的函数和短的函数，另外如果选择了正确的编译选项，还能使编译器生成易于调试的代码。）

9.2 有没有个简单的例子说明什么是顺序集成(procedure integration)？

考虑下面对函数 `g()` 的调用：

```
void f()
{
  int x = /*...*/;
  int y = /*...*/;
  int z = /*...*/;
  ...使用x, y 和 z的代码...
  g(x, y, z);
  ...更多使用x, y 和 z的代码...
}
```

假设一个典型的C++实现包含有一系列寄存器和一个栈，在调用 `g()` 之前，寄存器和参数会被写入栈中，然后在 `g()` 内部的栈中会读出参数值，然后在 `g()` 返回到 `f()` 时又会将这些寄存器的值读出来并恢复到寄存器中。但这里面有很多不必要的读写操作，尤其是当编译器能够用寄存器来保存 `x` 、 `y` 和 `z` 时。每 个变量都会写两次（做为寄存器和做为参数）并且读两次（在 `g()` 内部使用和返回到 `f()` 时恢复寄存器）。

```
void g(int x, int y, int z)
{
  ...使用x、 y 和 z的代码...
}
```

如果编译器能够内联展开对 `g()` 的调用，那么所有这些内存操作就都会消失了。不用再读写寄存器了，因为根本没有函数调用。各个参数也不必再被读写了，因为优化器知道它们已经在寄存器里了。

当然你所能获得的好处可能会变化，在本FAQ之外还有很多很多变数。但以上的例子能够揭示出顺序集成时所发生的事情。

9.3 内联函数能改善性能么？

可能会，也可能不会。有时可以。也许可以。

答案没那么简单。内联函数可能会使代码速度更快，也可能使速度变慢。可能会使可执行文件变大，也可能变小。可能会导致系统性能下降，也可能避免性能下降。内联函数可能（经常是）与速度完全无关。

`内联函数` 可能会使代码速度更快：正如上面所说，顺序集成可能会移除很多不必要的指令，这可能会加快速度。

`内联函数` 可能会使代码速度更慢：过多的内联可能会使代码膨胀，在使用分页虚拟内存的系统上，这可能会导致性能下降。换句话说，如果可执行文件过大，系统可能会花费很多时间到磁盘上获取下一块代码。

`内联函数` 可能会增加可执行文件尺寸：这就是上面所说的代码膨胀。例如，假设系统有100个内联函数，每个展开后有100字节，并且被调用了100次。这就会增加1MB的大小。增加这么1MB会导致问题吗？谁知道呢，但很可能就是这1MB导致系统性能下降。

`内联函数` 可能会减少可执行文件尺寸：如果不内联展开函数体，编译器可能会要产生更多代码来压入/弹出寄存器内容和参数。对于很小的函数来说会是这样。如果优化器能够通过顺序集成消除雕大量冗余代码的话，那么对大函数也会起作用（也就是说，优化器能够使大函数变小）。

`内联函数` 可能会导致系统性能下降：内联可能会导致二进制可执行文件尺寸变大，由此导致系统性能下降。

`内联函数`可能会避免系统性能下降：即使可执行文件尺寸变大，当前正在使用的物理内存数量（即需要同时留在内存中的页面数量）却仍然可能降低。当f()调用g()时，代码经常分散在2个不同的页面上。当编译器将g()的代码顺序集成到f()后，代码通常会放在一个页面上。

`内联函数`可能会降低缓存的命中率：内联可能会导致内层循环跨越多行的内存缓存，这可能会导致内存和缓存频繁交换，从而性能下降。

`内联函数`可能会提高缓存的命中率：内联通常能够在二进制代码中就近安排所用到的内容，这可能会减少用来存放内层循环代码的缓存数量。最终这会使CPU密集型程序跑得更快。

`内联函数`可能与速度无关：大多数系统不是CPU密集型的，而使I/O密集型的、数据库密集型的或是网 络密集型的。这表明系统的瓶颈存在于文件系统、数据库或网络。除非你的“CPU速度表”指示是100%，否则内联函数可能不会使你的系统速度更快。（即使 是CPU密集型的系统，也只有在被用到瓶颈之处时，内联才会有帮助。而瓶颈通常只存在于很少一部分代码中。）

没有简单定论：你需要多试验来找到最佳方案。不要指望依赖那些过分简化的答案，比如“绝不要使用内联函数”，或者“总是使用内联函数”，再比如“当且仅当函数体少于N行代码时使用内联函数”。这种以一盖全的准则写下来很容易，但却会产生不够优化的结果。

译注：这一小节中的性能下降主要是指系统因为频繁交换内存页而导致的性能下降。原文是thrashing。

9.4 内联函数如何在安全和速度上取得折衷？

在 C 中，你可以通过在结构中设置一个 `void*` 来得到“封装的结构”，在这种情况下，指向实际数据的 `void*` 指针对于结构的用户来说是未知的。因此结构的用户不知道如何解释 `void*` 指针所指内容，但是存取函数可以将 `void*` 转换成适当的隐含类型。这样给出了封装的一种形式。

不幸的是这样做丧失了类型安全，并且即使仅仅是访问结构体中的一个很不重要的字段也必须进行函数调用。（如果你允许直接存取结构的域，那么任何人都能直接存取该结构体了，因为他们必须了解如何解释 `void*` 指针所指内容；这样将使改变底层数据结构变的困难）。

虽然函数调用开销是很小的，但它会被累积。C++类允许函数调用以内联展开。这样让你在得到封装的安全性时，同时得到直接存取的速度。此外，内联函数的参数类型由编译器检查，这是对 C 的 `#define` 宏的一个改进。

9.5 为什么我应该用内联函数？而不是原来清晰的 #define 宏？

因为 `#define` 宏有四宗罪：罪状#1, 罪状#2, 罪状#3, 和 罪状#4。有时虽然你会用它们，但它们仍然是邪恶的。

和 #define 宏不同的是，内联函数总是对参数只精确地进行一次求值，从而避免了那声名狼藉的宏错误。换句话说，调用内联函数和调用正规函数是等价的，差别仅仅是更快：

```
// 返回 i 的绝对值的宏
#define unsafe(i)  \
        ( (i) >= 0 ? (i) : -(i) )

// 返回 i 的绝对值的内联函数
inline
int safe(int i)
{
  return i >= 0 ? i : -i;
}

int f();

void userCode(int x)
{
  int ans;

  ans = unsafe(x++);   // 错误！x 被增加两次
  ans = unsafe(f());   // 危险！f()被调用两次

  ans = safe(x++);     // 正确！ x 被增加一次
  ans = safe(f());     // 正确！ f() 被调用一次
}
```

和宏不同的，还有内联函数的参数类型被检查，并且被正确地进行必要的转换。

宏是有害的；非万不得已不要用。

9.6 如何告诉编译器使非成员函数成为内联函数？

声明内联函数看上去和普通函数非常相似：

```
void f(int i, char c);
```

当你定义一个内联函数时，在函数定义前加上 `inline` 关键字，并且将定义放入头文件：

```
inline
void f(int i, char c)
{
  // ...
}
```

注意：将函数的定义（`{` ... `}` 之间的部分）放在头文件中是强制的，除非该函数仅仅被单个 `.cpp` 文件使用。尤其是，如果你将内联函数的定义放在 `.cpp` 文件中并且在其他 `.cpp` 文件中调用它，连接器将给出 “unresolved external” 错误。

9.7 如何告诉编译器使一个成员函数成为内联函数？

声明内联成员函数看上去和普通成员函数非常类似：

```
class Fred {
public:
  void f(int i, char c);
};
```

但是当你定义内联成员函数时，在成员函数定义前加上 `inline` 关键字，并且将定义放入头文件中：

```
inline
void Fred::f(int i, char c)
{
  // ...
}
```

通常将函数的定义（`{` ... `}`之间的部分）放在头文件中是强制的，除非函数只在一个 `.cpp` 文件中用到。特别 是，如果你将内联函数的定义放在 `.cpp` 文件中并且在其他 `.cpp` 文件中调用它，连接器将给出 “unresolved external”错误。

9.8 有其它方法告诉编译器使成员函数成为内联吗？

有：在类体内定义成员函数：

```
class Fred {
public:
  void f(int i, char c)
    {
      // ...
    }
};
```

尽管这对于写类的人来说很容易，但由于它将类是“什么”(what)和类“如何”(how)工作混在一起，给阅读的人带来了困难。我们通常更愿意在类体外使用 `inline` 关键字定义成员函数来避免这种混合。这种感觉所基于的认识是：在一个面向重用的世界中，使用你的类的人有很多，而编写它的人只有一个（你自己）；因此你做任何事都应该照顾多数而不是少数。下一条FAQ进一步应用了这个方法。

9.9 在定义于类外部的内联函数中，以下哪种方法最好：是把 `inline` 关键字放在类内部的成员函数声明前呢，还是放到类外部函数的定义前呢，还是两个地方都写？

最佳实践是：仅放在类外部函数的定义前。

```
class Foo {
public:
  void method();  //← best practice: don't put the inline keyword here...
};

inline void Foo::method()  //← best practice: put the inline keyword here
{ ... }
```

这里是基本的想法：

- 类的 `public` 部分是你描述类的可见语义的地方，包含公有成员函数、友元函数和任何其它暴露给外部的内容。不要提供在调用者代码中看不到的细节。
- 类的其它部分，包括非公有部分、成员定义和友元函数声明等等，这些纯粹是实现细节。如果还没有在类的公有部分描述，那么不要提供相关可见语义。

从一种实际的观点来看，这种隔离能够使用户更轻松和更安全。假设 `Chuck` 只是想“用”你的类。因为你读了本FAQ并使用了上述隔离办 法， `Chuck` 能够在类的公有部分找到所有需要的内容，而不必看任何不需要的内容。他能够更加轻松，因为只需要看一个地方。同时也会更安全，因为他纯洁的 思想不必受到实现细节的干扰。

回到内联上来：一个函数是否内联只是实现细节，不会改变函数调用的可见语义（即含义）。因此 `inline` 关键字应该和函数定义放在一起，而不是在类的 `public` 声明区。

注意：大部分人使用“声明”和“定义”来区分以上所述的两个位置。例如，人们会说“我应该把 `inline` 关键字放到声明那里还是放在定义那里？”但 这种说法不太严密，可能会有人因此笑话你。笑话你的人可能只是不自信而又装腔作势的可怜虫，他们无法在其生命中取得一些成就。然而，你还是可以学会使用正 确的术语来避免被笑话。其实，每个定义同时也是声明。也就是说，如果把这两者当作是互斥的，那么就好像是在问钢和金属哪个更重。当你把“定义”说成是“声 明”的对立面时，几乎所有人都都明白你的意思。只有最糟糕的痴迷于技术的小人物才会因此嘲笑你。但至少你知道如何正确使用术语。

[10] 构造函数

FAQs in section [10]:

10.1 构造函数做什么？

构造函数从无到有创建对象。

构造函数就象“初始化函数”。它将一连串的随意的内存位变成活的对象。至少它要初始化对象内部所使用的域。它还可以分配资源（内存、文件、信号、套接字等）

"ctor" 是构造函数(constructor)典型的缩写。

10.2 `List x;` 和 `List x();` 有区别吗?

有非常大的区别！

假设 `List` 是某个类的名称。那么函数 `f()` 中声明了一个局部的 `List` 对象，名称为 `x` ：

```
void f()
{
  List x;     // Local object named x (of class List)// ...
}
```

但是函数 `g()` 中声明了一个名称为 `x()` 的函数，它返回一个 `List` ：

```
void g()
{
  List x();   // Function named x (that returns a List)// ...
}
```

10.3 如何才能够使一个构造函数直接地调用另一个构造函数？

不行。

注意：如果你调用了另一个构造函数，编译器将初始化一个临时局部对象；而不是初始化 `this` 对象。你可以通过一个默认参数或在一个私有成员函数 `init()` 中共享它们的公共代码来使两个构造函数结合起来。

10.4 `Fred` 类的默认构造函数总是 `Fred::Fred()` 吗？

不。“默认构造函数”是能够被无参数调用的构造函数。因此，一个不带参数的构造函数当然是默认构造函数：

```
class Fred {
public:
  Fred();   // 默认构造函数: 能够被无参数调用// ...
};
```

然而，如果参数被提供了默认值，那么带参数的默认构造函数也是可能的：

```
class Fred {
public:
  Fred(int i=3, int j=5);   // 默认构造函数: 能够被无参数调用// ...
};
```

10.5 当建立一个 `Fred` 对象数组时，哪个构造函数将被调用？

`Fred` 的默认构造函数（以下讨论除外）。

你无法告诉编译器调用不同的构造函数（以下讨论除外）。如果你的 `Fred` 类没有默认构造函数，那么试图创建一个 `Fred` 对象数组将会导致编译时出错。

```
class Fred {
public:
  Fred(int i, int j);
  // ... 假设 Fred 类没有默认构造函数 ...
};

int main()
{
  Fred a[10];               // 错误：Fred 类没有默认构造函数
  Fred* p = new Fred[10];   // 错误：Fred 类没有默认构造函数
}
```

然而，如果你正在创建一个标准的 `std::vector<Fred>` ，而不是 `Fred` 对象数组（既然数组是有害的，那么你可能应该这么做），则在 `Fred` 类中不需要默认构造函数。因为你能够给 `std::vector` 一个用来初始化元素的 `Fred` 对象：

```
#include <vector>

int main()
{
  std::vector<Fred> a(10, Fred(5,7));
  // 在std::vector 中的 10 个 Fred对象将使用 Fred(5,7) 来初始化// ...
}
```

虽然应该使用 `std::vector` 而不是数组，但有有应该使用数组的时候，那样的话，有“数组的显式初始化”语法。它看上去是这样的：

```
class Fred {
public:
  Fred(int i, int j);
  // ... 假设Fred类没有默认构造函数...
};

int main()
{
  Fred a[10] = {
    Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7),
    Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7)
  };

  // 10 个 Fred对象将使用 Fred(5,7) 来初始化.
  // ...
}
```

当然你不必每个项都做 `Fred(5,7)` —你可以放任何你想要的数字，甚至是参数或其他变量。重点是，这种语法是（a）可行的，但（b）不如 `std::vector` 语法漂亮。记住这个：数组是有害的—除非由于编译原因而使用数组，否则应该用 `std::vector` 取代。

10.6 构造函数应该用“初始化列表”还是“赋值”？

初始化列表。事实上，构造函数应该在初始化列表中初始化所有成员对象。

例如，构造函数用初始化列表`Fred::Fred() : x_(` *whatever* `) { }` 来初始化成员对象 `x_` 。这样做最普通的好处是提高性能。如，*whatever*表达式和成员变量 `x_` 相同，*whatever*表达式的结果直接由内部的`x_`来构造——编译器不会产生对象的两个拷贝。即使类型不同，使用初始化列表时编译器通常也能够做得比使用赋值更好。

建立构造函数的另一种（错误的）方法是通过赋值，如：`Fred::Fred() { x_ =` *whatever* `; }` 。在这种情况下，*whatever*表达式导致一个分离的，临时的对象被建立，并且该临时对象被传递给`x_`对象的赋值操作。然后该临时对象会在 ；处被析构。这样是效率低下的。

这好像还不是太坏，但这里还有一个在构造函数中使用赋值的效率低下之源：成员对象会被以默认构造函数完整的构造，例如，可能分配一些缺省数量的内存或打开一些缺省的文件。但如果 *whatever*表达式和／或赋值操作导致对象关闭那个文件和／或释放那块内存，这些工作是做无用功（举例来说，如默认构造函数没有分配一个足够大的内存池或它打开了错误的文件）。

结论：其他条件相等的情况下，使用初始化列表的代码会快于使用赋值的代码。

注意：如果`x_`的类型是诸如`int`或者`char*` 或者`float`之类的内建类型，那么性能是没有区别的。但即使在这些情况下，我个人的偏好是为了对称，仍然使用初始化列表而不是赋值来设置这些数据成员。

10.7 可以在构造函数中使用 `this` 指针吗？

某些人认为不应该在构造函数中使用`this`指针，因为这时`this`对象还没有完全形成。然后，只要你小心，是可以在构造函数（在函数体甚至在初始化列表中）使用`this`的。

以下是始终可行的：构造函数的函数体（或构造函数所调用的函数）能可靠地访问基类中声明的数据成员和／或构造函数所属类声明的数据成员。这是因为所有这些数据成员被保证在构造函数函数体开始执行时已经被完整的建立。

以下是始终不可行的：构造函数的函数体（或构造函数所调用的函数）不能向下调用被派生类 重定义的虚函数。如果你的目的是得到派生类重定义的函数，那么你将无功而返。注意，无论你如何调用虚成员函数：显式使用`this`指针（如，`this->method()`），隐式的使用`this`指针（如，`method()`），或甚至在`this`对象上调用其他函数来调用该虚成员函数，你都不会得到派生类的重写函数。这是底线：即使调用者正在构建一个派生类的对象，在基类的构造函数执行期间，对象还不是一个派生类的对象。

以下是有时可行的：如果传递 `this` 对象的任何一个数据成员给另一个数据成员的初始化程序，你必须确保该数据成员已经被初始化。好消息是你能使用一些不依赖于你所使用的编译器的显著的语言规则，来确定那个数据成员是否已经（或者还没有）被初始化。坏消息是你必须知道这些语言规则（例如，基类子对象首先被初始化（如果有多重和／或虚继承，则查

询这个次序！），然后类中定义的数据成员根据在类中声明的次序被初始化）。如果你不知道这些规则，则不要从 `this` 对象传递任何数据成员（不论是否显式的使用了 `this` 关键字）给任何其他数据成员的初始化程序！如果你知道这些规则，则需要小心。

10.8 什么是“命名的构造函数法（Named Constructor Idiom）”？

为你的类的用户提供的一种更直觉的和／或更安全的构造操作技巧。

问题在于构造函数总是有和类相同的名字。因此，区分类的不同的构造函数是通过参数列表。但如果有许多构造函数，它们之间的区别有时就会很敏感并且有错误倾向。

使用命名的构造函数法（Named Constructor Idiom），在 `private:` 节和 `protected:` 节中声明所有类的构造函数，并提供返回一个对象的 `public` `static` 方法。这些方法由此称为“命名的构造函数（Named Constructors）”。一般，每种不同的构造对象的方法都有一个这样的静态方法。

例如，假设我们正在建立一个描绘X-Y平面的 `Point` 类。通常有两种方法指定一个二维空间坐标：矩形坐标(X+Y)，极坐标(Radius+Angle)（半径＋角度）。（不必担心已经忘了这些；重点不在于坐标系统的析解；重点在于有几种方法来创建一个 `Point` 对象。）不幸的是，这两种坐标系统的参数是相同的：两个 `float` 。这将在重载构造函数中导致一个“重载不明确”的错误：

```
class Point {
public:
  Point(float x, float y);     // 矩形坐标_
  Point(float r, float a);     // 极坐标 (半径和角度)
  // 错误：重载不明确：Point::Point(float,float)
};

int main()
{
  Point p = Point(5.7, 1.2);   // 不明确：哪个坐标系统？
}
```

解决这个不明确错误的一种方法是使用命名的构造函数法（Named Constructor Idiom）：

```
#include <cmath>                // To get sin() and cos()

class Point {
public:
  static Point rectangular(float x, float y);      // 矩形坐标_
  static Point polar(float radius, float angle);   // 极坐标
  // 这些 static 方法称为“命名的构造函数（named constructors）”
  // ...
private:
  Point(float x, float y);     // 矩形坐标
  float x_, y_;
};

inline Point::Point(float x, float y)
: x_(x), y_(y) { }

inline Point Point::rectangular(float x, float y)
{ return Point(x, y); }

inline Point Point::polar(float radius, float angle)
{ return Point(radius*cos(angle), radius*sin(angle)); }
```

现在，`Point`的用户有了一个清晰的和明确的语法在任何一个坐标系统中创建`Point`对象：

```
int main()
{
  Point p1 = Point::rectangular(5.7, 1.2);   // 显然是矩形坐标
  Point p2 = Point::polar(5.7, 1.2);         // 显然是极坐标
}
```

如果期望`Point`有派生类，则确保你的构造函数在`protected:`节中。

命名的构造函数法也能用于总是通过`new`来创建对象。

10.9 为何不能在构造函数的初始化列表中初始化静态成员数据？

因为必须显式定义类的静态数据成员。

```
//Fred.h:

 class Fred {
 public:
   Fred();
   // ...
 private:
   int i_;
   static int j_;
 };


//Fred.cpp (或 Fred.C 或其他):

 Fred::Fred()
   : i_(10)  // 正确：能够（而且应该）这样初始化成员数据
   , j_(42)  // 错误：不能象这样初始化静态成员数据
 {
   // ...
 }

 // 必须这样定义静态数据成员：
 int Fred::j_ = 42;
```

10.10 为何有静态数据成员的类得到了链接错误？

因为静态数据成员必须被显式定义在一个编辑单元中。如果不这样做，你就可能得到 `"undefined external"` 链接错误。例如：

```
 // Fred.h

 class Fred {
 public:
   // ...
 private:
   static int j_;   // 声明静态数据成员：Fred::j_
   // ...
 };
```

链接器会向你抱怨（ `"Fred::j_ is not defined"` ），除非你在一个源文件中定义（而不仅仅是声明） `Fred::j_` ：

```
 // Fred.cpp

 #include "Fred.h"

 int Fred::j_ = some_expression_evaluating_to_an_int;

 // Alternatively, if you wish to use the implicit 0 value for static ints:
 // int Fred::j_;
```

通常定义 `Fred` 类的静态数据成员的地方是 `Fred.cpp` 文件（或者 `Fred.C` 或者你使用的其他扩展名）。

10.11 什么是“ `static` initialization order fiasco”？

你的项目的微妙杀手。

`static` *initialization order fiasco*是对C++的一个非常微妙的并且常见的误解。不幸的是，错误发生在`main()`开始之前，很难检测到。

简而言之，假设你有存在于不同的源文件`x.cpp` 和`y.cpp`的两个静态对象`x` 和 `y` 。再假定`y`对象的构造函数会调用`x`对象的某些方法。

就是这些。就这么简单。

结局是你完蛋不完蛋的机会是50%-50%。如果碰巧`x.cpp`的编辑单元先被初始化，这很好。但如果`y.cpp`的编辑单元先被初始化，然后`y`的构造函数比`x`的构造函数先运行。也就是说，`y`的构造函数会调用`x`对象的方法，而`x`对象还没有被构造。

我听说有些人受雇于麦当劳，享受他们的切碎肉的新工作去了。

如果你觉得不用工作，在卧室的一角玩俄罗斯方块是令人兴奋的，你可以到此为止。相反，如果你想通过用一种系统的方法防止灾难，来提高自己 继续工作而存活的机会，你可能想阅读下一个 FAQ。

注意：static initialization order fiasco不作用于内建的／固有的类型，象`int` 或 `char*` 。例如，如果创建一个`static` `float`对象，不会有静态初始化次序的问题。静态初始化次序真正会崩溃的时机只有在你的`static`或全局对象有构造函数时。

10.12 如何防止“`static` initialization order fiasco”？

使用“首次使用时构造（construct on first use）”法，意思就是简单地将静态对象包裹于函数内部。

例如，假设你有两个类，`Fred` 和 `Barney` 。有一个称为`x`的全局`Fred`对象，和一个称为`y`的全局`Barney`对象。`Barney`的构造函数调用了`x`对象的`goBowling()`方法。`x.cpp`文件定义了`x`对象：

```
// File x.cpp
#include "Fred.hpp"
Fred x;
```

`y.cpp`文件定义了`y`对象：

```
// File y.cpp
#include "Barney.hpp"
Barney y;
```

`Barney`构造函数的全部看起来可能是象这样的：

```
// File Barney.cpp
#include "Barney.hpp"

Barney::Barney()
{
  // ...
  x.goBowling();
  // ...
}
```

正如以上所描述的，由于它们位于不同的源文件，那么 `y` 在 `x` 之前构造而发生灾难的机率是50%。

这个问题有许多解决方案，但一个非常简便的方案就是用一个返回 `Fred` 对象引用的全局函数 `x()` ，来取代全局的 `Fred` 对象 `x` 。

```
// File x.cpp

#include "Fred.hpp"

Fred& x()
{
  static Fred* ans = new Fred();
  return *ans;
}
```

由于静态局部对象只在控制流第一次越过它们的声明时构造，因此以上的 `new Fred()` 语句只会执行一次： `x()` 被第一次调用时。每个后续的调用将返回同一个 `Fred` 对象（ `ans` 指向的那个）。然后你所要做的就是将 `x` 改成 `x()` ：

```
// File Barney.cpp
#include "Barney.hpp"

Barney::Barney()
{
  // ...
  x().goBowling();
  // ...
}
```

由于该全局的 `Fred` 对象在首次使用时被构造，因此被称为首次使用时构造法（*Construct On First Use Idiom*）

这种方法的不利方面是 `Fred` 对象不会被析构。*C++ FAQ Book*有另一种技巧消除这个影响（但面临了“static *de*-initialization order fiasco”的代价）。

注意：对于内建／固有类型，象 `int` 或 `char*` ，不必这样做。例如，如果创建一个静态的或全局的 `float` 对象，不需要将它包裹于函数之中。静态初始化次序真正会崩溃的时机只有在你的 `static` 或全局对象有构造函数时。

10.13 对于静态数据成员，如何防止“`static initialization order fiasco`”？

使用与描述过的相同的技巧，但这次使用静态成员函数而不是全局函数而已。

假设类 `X` 有一个`static` `Fred` 对象：

```
// File X.hpp

class X {
public:
  // ...

private:
  static Fred x_;
};
```

自然的，该静态成员被分开初始化：

```
// File X.cpp

#include "X.hpp"

Fred X::x_;
```

自然的，`Fred` 对象会在 `X` 的一个或多个方法中被使用：

```
void X::someMethod()
{
  x_.goBowling();
}
```

但现在“灾难情景”就是如果某人在某处不知何故在`Fred` 对象被构造前调用这个方法。例如，如果某人在静态初始化期间创建一个静态的 `X` 对象并调用它的`someMethod()` 方法，然后你就受制于编译器是在`someMethod()` 被调用之前或之后构造 `X::x_` 。（ANSI/ISO C++委员会正在设法解决这个问题，但诸多的编译器对处理这些更改一般还没有完成；关注此处将来的更新。）

无论何种结果，将`X::x_` 静态数据成员改为静态成员函数总是最简便和安全的：

```
// File X.hpp

class X {
public:
  // ...

private:
  static Fred& x();
};
```

自然的，该静态成员被分开初始化：

```
// File X.cpp

#include "X.hpp"

Fred& X::x()
{
  static Fred* ans = new Fred();
  return *ans;
}
```

然后，简单地将 `x_` 改为 `x()` ：

```
void X::someMethod()
{
  x().goBowling();
}
```

如果你对性能敏感并且关心每次调用 `X::someMethod()` 的额外的函数调用的开销，你可以设置一个 `static` `Fred&` 来取代。正如你所记得的，静态局部对象仅被初始化一次（控制流程首次越过它们的声明处时），因此，将只调用 `X::x()` 一次： `X::someMethod()` 首次被调用时：

```
void X::someMethod()
{
  static Fred& x = X::x();
  x.goBowling();
}
```

注意：对于内建／固有类型，象 `int` 或 `char*` ，不必这样做。例如，如果创建一个静态的或全局的 `float` 对象，不需要将它包裹于函数之中。静态初始化次序真正会崩溃的时机只有在你的 `static` 或全局对象有构造函数时。

10.14 如何处理构造函数的失败？

抛出一个异常。详见 [17.2]。

10.15 什么是“命名参数法（Named Parameter Idiom）”？

发掘方法链的非常有用的方法。

命名参数法（Named Parameter Idiom）解决的最基本问题是C++仅支持位置相关的参数。例如，函数调用者不能说“这个值给形参 `xyz` ，另一个值给形参 `pqr` ”。在C++（和C 和Java）中只能说“这是第一个参数，这是第二个参数等”。Ada语言提出并实现的命名参数，对于带有大量的可缺省参数的函数尤其有用。

多年来，人们构造了很多方案来弥补C 和 C++缺乏的命名参数。其中包括将参数值隐藏于一个字符串参数，然后在运行时解析这个字符串。例如，这就是 `fopen()` 的第二个参数的做法。另一种方案是将所有的布尔参数联合成一个位映射，然后调用者将这堆转换成位的常量共同产生一个实际的参数。例如，这就是 `open()` 的第二个参数的做法。这些方法可以工作，但下面的技术产生的调用者的代码更明显，更容易写，更容易读，而且一般来说更雅致。

这个想法，称为命名参数法（Named Parameter Idiom），它是将函数的参数变为以新的方式创建的类的方法，这些方法通过引用返回 `*this` 。然后你只要将主要的函数改名为那个类中的无参数的“随意”方法。

举一个例子来解释上面那段。

这个例子实现“打开一个文件”的概念。该概念逻辑上需要一个文件名的参数，和一些允许选择的参数，文件是否被只读或可读写或只写的方式打开；如果文件不存在，是否创建它；是从末尾写（添加"append"）还是从起始处写（覆盖"overwrite"）；如果文件被创建，指定块大小； I/O是否有缓冲区，缓冲区大小；文件是被共享还是独占访问；以及其他可能的选项。如果我们用常规的位置相关的参数的函数实现这个概念，那么调用者的代码会非常难读：有8个可选的参数，并且调用者很可能犯错误。因此我们使用命名参数用法来取代。

在实现它之前，假如你想接受函数的所有默认参数，看一下调用者的代码是什么样子：

```
File f = OpenFile("foo.txt");
```

那是简单的情况。现在看一下如果你想改变一大堆的参数：

```
 File f = OpenFile("foo.txt").
            readonly().
            createIfNotExist().
            appendWhenWriting().
            blockSize(1024).
            unbuffered().
            exclusiveAccess();
```

注意这些“参数”，被公平的以随机的顺序（位置无关的）调用并且都有名字。因此，程序员不必记住参数的顺序，而且这些名字是（正如所希望的）意义明显的。

以下是如何实现：首先创建一个新的类(`OpenFile`)，该类包含了所有的参数值作为 `private:` 数据成员。然后所有的方法（ `readonly()` , `blockSize(unsigned)` , 等）返回 `*this` （也就是返回一个 `OpenFile` 对象的引用，以允许方法被链状调用）。最后完成一个带有必要参数（在这里，就是文件名）的常规的，参数位置相关的 `OpenFile` 的构造函数。

```
class File;

class OpenFile {
public:
  OpenFile(const string& filename);
  // 为每个数据成员设置默认值
  OpenFile& readonly();  // 将 readonly_ 变为 true
  OpenFile& createIfNotExist();
  OpenFile& blockSize(unsigned nbytes);
  // ...
private:
  friend File;
  bool readonly_;       // 默认为 false [举例]
  // ...
  unsigned blockSize_;  // 默认为 4096 [举例]
  // ...
};
```

要做的另外一件事就是使得 `File` 的构造函数带一个`OpenFile`对象：

```
class File {
public:
  File(const OpenFile& params);
  // vacuums the actual params out of the OpenFile object
  // ...
};
```

注意`OpenFile` 将 `File` 声明为友元。

[11] 析构函数

FAQs in section [11]:

- [11.1] 析构函数做什么？
- [11.2] 局部对象析构的顺序是什么？
- [11.3] 数组中的对象析构顺序是什么？
- [11.4] 我能重载类的析构函数吗？
- [11.5] 我可以对局部变量显式调用析构函数吗？
- [11.6] 如果我要一个局部对象在其被创建的代码块的 } 之前被析构，如果我眞的想这样，能调用其析构函数吗？
- [11.7] 好，好；我不显式调用局部对象的析构函数；但如何处理上面那种情况？
- [11.8] 如果我无法将局部对象包裹于人为的块中，怎么办？
- [11.9] 如果我是用 `new` 分配对象的，可以显式调用析构函数吗？
- [11.10] 什么是“定位放置 `new` （placement `new` ）”，为什么要用它？
- [11.11] 编写析构函数时，需要显式调用成员对象的析构函数吗？
- [11.12] 当我写派生类的析构函数时，需要显式调用基类的析构函数吗？
- [11.13] 当析构函数检测到错误时，可以抛出异常吗？

11.1 析构函数做什么？

析构函数为对象举行葬礼。

析构函数用来释放对象所分配的资源。举例来说， `Lock` 类可能锁定了一个信号量，那么析构函数将释放该信号量。最常见的例子是，当构造函数中使用了 `new` ，那么析构函数则使用 `delete` 。

析构函数是“准备后事”的成员函数。经常缩写成“dtor”。

11.2 局部对象析构的顺序是什么？

与构造函数反序：先被构造的，被后析构。

以下的例子中， `b` 的析构函数会被首先执行，然后是 `a` 的析构函数：

```
void userCode()
{
  Fred a;
  Fred b;
  // ...
}
```

11.3 数组中的对象析构顺序是什么？

与构造函数反序：先被构造的，后被析构。

以下的例子中，析构的顺序是 `a[9]` , `a[8]` , ..., `a[1]` , `a[0]` ：

```
void userCode()
{
  Fred a[10];
  _// ..._
}
```

11.4 我能重载类的析构函数吗？

不行。

类只能有一个析构函数。 `Fred` 类的析构函数能是 `Fred::~Fred()` 。不带任何参数，不返回任何东西（译注：void也不行）。

由于你不会显式地调用析构函数（是的，永远不会），因此无论如何不能传递参数给析构函数。

11.5 我可以对局部变量显式调用析构函数吗？

不行！

在创建该局部对象的代码块的 `}` 处，析构函数会自动被调用。这是语言所保证的；自动发生。没有办法阻止它。而两次调用同一个对象的析构函数，你得到的真是坏的结果！砰！你完蛋了！

11.6 如果我要一个局部对象在其被创建的代码块的 `}` 之前被析构，如果我真的想这样，能调用其析构函数吗？

不行！详见 [前一个FAQ].

假设析构 `File` 对象的作用是关闭文件。现在假定你有一个 `File` 类的对象 `f` ，并且你想 `File` `f` 在 `f` 对象的作用范围结束（也就是 `}` ）之前被关闭：

```
void someCode()
{
  File f;

  // ... [这些代码在 f 打开的时候执行] ...
  // <— 希望在此处关闭 f
  // ... [这些代码在 f 关闭后执行] ...
}
```

对这个问题有一个简单的解决方案。但现在请记住：不要显式调用析构函数！

11.7 好，好；我不显式调用局部对象的析构函数；但如何处理上面那种情况？

内容详见 [前一个 FAQ].

只要将局部对象的生命期长度包裹于一个人为的 `{` ... `}` 块中：

```
void someCode()
{
  {
    File f;
    // ... [这些代码在 f 打开的时候执行] ...
  }
  // ^— f 的析构函数在此处会被自动调用！
  // ... [这些代码在 f 关闭后执行] ...
}
```

11.8 如果我无法将局部对象包裹于人为的块中，怎么办？

大多数时候，你可以通过将局部对象包裹于人为的 `{` ... `}` 块中，限制其生命期。但如果由于一些原因无法这样做，则增加一个模拟析构函数作用的成员函数。但不要调用析构函数本身！

例如， `File` 类的情况下，可以添加一个 `close()` 方法。典型的析构函数只是调用 `close()` 方法。注意 `close()` 方法需要标记 `File` 对象，以便后续的调用不会再次关闭一个已经关闭的文件。举例来说，可以将一个 `fileHandle_` 数据成员设置为 -1，并且在开头检查 `fileHandle_` 是否已经等于-1：

```
class File {
public:
  void close();
  ~File();
  // ...
private:
  int fileHandle_;   // 当且仅当文件打开时 fileHandle_ >= 0
};

File::~File()
{
  close();
}

void File::close()
{
  if (fileHandle_ >= 0) {
    // ... [执行一些操作－系统调用来关闭文件] ...
    fileHandle_ = -1;
  }
}
```

注意其他的 `File` 方法可能也需要检查`fileHandle_`是否为 -1（也就是说，检查文件是否被关闭了）。

还要注意任何没有实际打开文件的构造函数，都应该将`fileHandle_`设置为 -1。

11.9 如果我是用 `new` 分配对象的，可以显式调用析构函数吗？

可能不行。

除非你使用定位放置 `new`，否则应该 `delete` 对象而不是显式调用析构函数。例如，假设通过一个典型的 `new` 表达式分配一个对象：

```
Fred* p = new Fred();
```

那么，当你`delete`它时，析构函数 `Fred::~Fred()` 会被调用：

```
delete p;  // 自动调用 p->~Fred()
```

由于显式调用析构函数不会释放 `Fred` 对象本身分配的内存，因此不要这样做。记住：`delete p` 做了两件事情：调用析构函数，回收内存。

11.10 什么是“定位放置 `new` （placement `new` ）”，为什么要用它 ？

定位放置`new`（placement `new`）有很多作用。最简单的用处就是将对象放置在内存中的特殊位置。这是依靠 `new`表达式部分的指针参数的位置来完成的：

```
#include <new>        // 必须 #include 这个，才能使用 "placement new"
#include "Fred.h"     // class Fred 的声明

void someCode()
{
  char memory[sizeof(Fred)];     // Line #1
  void* place = memory;          // Line #2

  Fred* f = new(place) Fred();   // Line #3 (详见以下的“危险”)
  // The pointers f and place will be equal
  // ...
}
```

Line #1 在内存中创建了一个`sizeof(Fred)`字节大小的数组，足够放下 `Fred` 对象。Line #2 创建了一个指向这块内存的首字节的`place`指针（有经验的 C 程序员会注意到这一步是多余的，这儿只是为了使代码更明显）。Line #3 本质上只是调用了构造函数 `Fred::Fred()` 。`Fred`构造函数中的`this`指针将等于`place` 。因此返回的 `f` 将等于`place` 。

建议：万不得已时才使用“placement `new`”语法。只有当你真的在意对象在内存中的特定位置时才使用它。例如，你的硬件有一个内存映象的 I/O计时器设备，并且你想放置一个`Clock`对象在那个内存位置。

危险：你要独自承担这样的责任，传递给“placement `new`”操作符的指针所指向的内存区域必须足够大，并且可能需要为所创建的对象进行边界调整。编译器和运行时系统都不会进行任何的尝试来检查你做的是否正确。如果 `Fred` 类需要将边界调整为4字节，而你提供的位置没有进行边界调整的话，你就会亲手制造一个严重的灾难（如果你不明白“边界调整”的意思，那么就不要使用placement `new` 语法）。

你还有析构放置的对象的责任。这通过显式调用析构函数来完成：

```
void someCode()
{
  char memory[sizeof(Fred)];
  void* p = memory;
  Fred* f = new(p) Fred();
  // ...
  f->~Fred();   // 显式调用定位放置的对象的析构函数
}
```

这是显式调用析构函数的唯一时机。

11.11 编写析构函数时，需要显式调用成员对象的析构函数吗？

不！永远不需要显式调用析构函数（除了定位放置 `new` 的情况）。

类的析构函数（不论你是否显式地定义了）自动调用成员对象的析构函数。它们以出现在类声明中的顺序的反序被析构。

```
class Member {
public:
  ~Member();
  // ...
};

class Fred {
public:
  ~Fred();
  _// ..._
private:
  Member x_;
  Member y_;
  Member z_;
};

Fred::~Fred()
{
  // 编译器自动调用 z_.~Member()
  // 编译器自动调用 y_.~Member()
  // 编译器自动调用 x_.~Member()
}
```

11.12 当我写派生类的析构函数时，需要显式调用基类的析构函数吗？

不！永远不需要显式调用析构函数（除了定位放置 `new` 的情况）。

派生类的析构函数（不论你是否显式地定义了）自动调用基类子对象的析构函数。基类在成员对象之后被析构。在多重继承的情况下，直接基类以出现在继承列表中的顺序的反序被析构。

```
class Member {
public:
  ~Member();
  // ...
};

class Base {
public:
  virtual ~Base();     // 虚析构函数
  // ...
};

class Derived : public Base {
public:
  ~Derived();
  // ...
private:
  Member x_;
};

Derived::~Derived()
{
  // 编译器自动调用 x_.~Member()
  // 编译器自动调用 Base::~Base()
}
```

注意：虚拟继承的顺序相关性是多变的。如果你在一个虚拟继承层次中依赖于其顺序相关性，那么你需要比这个FAQ更多的信息。

11.13 当析构函数检测到错误时，可以抛出异常吗？

谨防!!! 详见 该 FAQ。

[12] 赋值算符

FAQs in section [12]:

12.1 什么是“自赋值”？

自赋值就是将对象赋值给本身。例如，

```
#include "Fred.hpp"    // 声明 Fred 类

void userCode(Fred& x)
{
  x = x;   // 自赋值
}
```

很明显，以上代码进行了显式的自赋值。但既然多个指针或引用可以指向相同对象（别名），那么进行了自赋值而自己却不知道的情况也是可能的：

```
#include "Fred.hpp"    // 声明 Fred 类

void userCode(Fred& x, Fred& y)
{
  x = y;   // 如果&x == &y就可能是自赋值
}

int main()
{
  Fred z;
  userCode(z, z);
}
```

12.2 为什么应该当心“自赋值”？

如果不注意自赋值，将会使你的用户遭受非常微妙的并且一般来说非常严重的bug。例如，如下的类在自赋值的情况下将导致灾难：

```
class Wilma { };

class Fred {
public:
  Fred()                : p_(new Wilma())      { }
  Fred(const Fred& f)   : p_(new Wilma(*f.p_)) { }
 ~Fred()                { delete p_; }
  Fred& operator= (const Fred& f)
    {
      // 差劲的代码：没有处理自赋值！
      delete p_;                // Line #1
      p_ = new Wilma(*f.p_);    // Line #2
      return *this;
    }
private:
  Wilma* p_;
};
```

如果有人将 `Fred` 对象赋给其本身，由于 `*this` 和 `f` 是同一个对象，line #1同时删除了 `this->p_` 和 `f.p_` 。而 line #2使用了已经不存在的对象 `*f.p_` ，这样很可能导致严重的灾难。

作为 `Fred` 类的作者，你最起码有责任确信在 `Fred` 对象上自赋值是无害的。不要假设用户不会在对象上这样做。如果对象由于自赋值而崩溃，那是你的过失。

> 另外：上述的 `Fred::operator= (const Fred&)` 还有第二个问题：如果在执行 `new Wilma(*f.p_)` 时，抛出了异常或者 `Wilma` 的拷贝构造函数中的异常），`this->p_` 将成为悬空指针——它所指向的内存不再是可用的。这可以通过在删除就对象前创建对象来解决。

12.3 好，好；我会处理自赋值的。但如何做呢？

在你创建类的每时每刻，都应该当心自赋值。这并不意味着需要为你所有的类都增加额外的代码：只要对象优雅地处理自赋值，而不管是否必须增加额外的代码。

如果不需要为赋值算符增加额外代码，这里有一个简单而有效的技巧：

```
Fred& Fred::operator= (const Fred& f)
{
  if (this == &f) return *this;   // 优雅地处理自赋值
  // 此处写正常赋值的代码...

  return *this;
}
```

显式的测试并不总是必要的。例如，如果修正前一个 FAQ中的赋值算符使之处理 `new` 抛出的异常和／或 `Wilma` 类的拷贝构造函数抛出的异常，可能会写出如下的代码。注意这段代码有（令人高兴的）自动处理自赋值的附带效果：

```
Fred& Fred::operator= (const Fred& f)
{
  // 这段代码优雅地（但隐含的）处理自赋值
  Wilma* tmp = new Wilma(*f.p_);   // 如果异常在此处被抛出也没有问题
  delete p_;
  p_ = tmp;
  return *this;
}
```

在象这个例子的情况下（自赋值是无害的但是低效），一些程序员想通过增加另外的不必要的测试，如“`if (this == &f) return *this;`”来改善自赋值时的效率。通常来说，使自赋值情况更高效而使得非自赋值情况更低效的折衷是错误的。例如，为 `Fred` 类的赋值算符增加如上的 `if` 测试会使得非自赋值情况更低效（一个额外的(而且不必要的)条件分支）。如果自赋值实际上一千次才发生一次，那么 `if` 将浪费99.9%的时间周期。

[13] 运算符重载

FAQs in section [13]:

13.1 运算符重载的作用是什么？

它允许你为类的用户提供一个直觉的接口。

运算符重载允许C/C++的运算符在用户定义类型（类）上拥有一个用户定义的意义。重载的运算符是函数调用的语法修饰：

```
class Fred {
public:
  // ...
};

#if 0

  // 没有运算符重载：
  Fred add(Fred, Fred);
  Fred mul(Fred, Fred);

  Fred f(Fred a, Fred b, Fred c)
  {
    return add(add(mul(a,b), mul(b,c)), mul(c,a));    // 哈哈，多可笑...
  }

#else

  // 有运算符重载：
  Fred operator+ (Fred, Fred);
  Fred operator* (Fred, Fred);

  Fred f(Fred a, Fred b, Fred c)
  {
    return a*b + b*c + c*a;
  }

#endif
```

13.2 运算符重载的好处是什么？

通过重载类上的标准运算符，你可以发掘类的用户的直觉。使得用户程序所用的语言是面向问题的，而不是面向机器的。

最终目标是降低学习曲线并减少错误率。

13.3 有什么运算符重载的实例？

这里有一些运算符重载的实例：

- `myString + yourString` 可以连接两个 `std::string` 对象
- `myDate++` 可以增加一个 `Date` 对象
- `a * b` 可以将两个 `Number` 对象相乘
- `a[i]` 可以访问 `Array` 对象的某个元素
- `x = *p` 可以反引用一个实际“指向”一个磁盘记录的 "smart pointer" —— 它实际上在磁盘上定位到 `p` 所指向的记录并返回给 `x` 。

13.4 但是运算符重载使得我的类很丑陋；难道它不是应该使我的类更清晰吗？

运算符重载使得类的用户的工作更简易，而不是为类的开发者服务的！

考虑一下如下的例子：

```
class Array {
public:
  int& operator[] (unsigned i);      // 有些人不喜欢这种语法
  // ...
};

inline
int& Array::operator[] (unsigned i)  // 有些人不喜欢这种语法
{
  // ...
}
```

有些人不喜欢 `operator` 关键字或类体内的有些古怪的语法。但是运算符重载语法不是被期望用来使得类的开发者的工作更简易。它被期望用来使得类的用户的工作更简易：

```
int main()
{
  Array a;
  a[3] = 4;   // 用户代码应该明显而且易懂...
}
```

记住:在一个面向重用的世界中，使用你的类的人有很多，而建造它的人只有一个（你自己）；因此你做任何事都应该照顾多数而不是少数。

13.5 什么运算符能／不能被重载？

大多数都可以被重载。C的运算符中只有 `.` 和 `? :` （以及`sizeof`，技术上可以看作一个运算符）。C++增加了一些自己的运算符，除了`::`和`.*`，大多数都可以被重载。

这是一个下标 运算符的示例（它返回一个引用）。先没有运算符重载：

```
class Array {
public:
  int& elem(unsigned i)        { if (i > 99) error(); return data[i]; }
private:
  int data[100];
};

int main()
{
  Array a;
  a.elem(10) = 42;
  a.elem(12) += a.elem(13);
}
```

现在用运算符重载给出同样的逻辑：

```
class Array {
public:
  int& operator[] (unsigned i) { if (i > 99) error(); return data[i]; }
private:
  int data[100];
};

int main()
{
  Array a;
  a[10] = 42;
  a[12] += a[13];
}
```

13.6 我能重载 `operator==` 以便比较两个 `char[]` 来进行字符串比较吗？

不行：被重载的运算符，至少一个操作数必须是用户定义类型（大多数时候是类）。

但即使C++允许，也不要这样做。因为在此处你应该使用类似 `std::string` 的类而不是字符数组，因为数组是有害的。因此无论如何你都不会想那样做的。

13.7 我能为“幂”运算创建一个 `operator**` 吗？

不行。

运算符的名称、优先级、结合性以及元数都是由语言固定的。在C++中没有`operator**`，因此你不能为类类型创建它。

如果还有疑问，考虑一下`x ** y`与`x * (*y)`等同（换句话说，编译器假定 `y` 是一个指针）。此外，运算符重载只不过是函数调用的语法修饰。虽然这种特殊的语法修饰非常美妙，但它没有增加任何本质的东西。我建议你重载`pow(base,exponent)`（双精度版本在`<cmath>`中）。

顺便提一下，`operator^`可以成为幂运算，只是优先级和结合性是错误的。

13.8 如何为 `Matrix` （矩阵）类创建下标运算符？

用 `operator()`而不是`operator[]`。

当有多个下标时,最清晰的方式是使用`operator()`而不是`operator[]`。原因是`operator[]`总是带一个参数，而`operator()`可以带任何数目的参数（在矩形的矩阵情况下，需要两个参数）。

如：

```
class Matrix {
public:
  Matrix(unsigned rows, unsigned cols);
  double& operator() (unsigned row, unsigned col);
  double  operator() (unsigned row, unsigned col) const;
  _// ..._
 ~Matrix();                              // 析构函数
  Matrix(const Matrix& m);               // 拷贝构造函数
  Matrix& operator= (const Matrix& m);   // 赋值运算符
  // ...
private:
  unsigned rows_, cols_;
  double* data_;
};

inline
Matrix::Matrix(unsigned rows, unsigned cols)
  : rows_ (rows),
    cols_ (cols),
    data_ (new double[rows * cols])
{
  if (rows == 0 || cols == 0)
    throw BadIndex("Matrix constructor has 0 size");
}

inline
Matrix::~Matrix()
{
  delete[] data_;
}

inline
double& Matrix::operator() (unsigned row, unsigned col)
{
  if (row >= rows_ || col >= cols_)
    throw BadIndex("Matrix subscript out of bounds");
  return data_[cols_*row + col];
}

inline
double Matrix::operator() (unsigned row, unsigned col) const
{
  if (row >= rows_ || col >= cols_)
    throw BadIndex("const Matrix subscript out of bounds");
  return data_[cols_*row + col];
}
```

然后，你可以使用`m(i,j)`来访问`Matrix` `m` 的元素，而不是`m[i][j]`：

```
int main()
{
  Matrix m(10,10);
  m(5,8) = 106.15;
  std::cout << m(5,8);
  // ...
}
```

13.9 为什么 `Matrix` （矩阵）类的接口不应该象数组的数组？

本 FAQ 其实是关于：某些人建立的Matrix 类，带有一个返回 `Array` 对象的引用的 `operator]` 。而该 `Array` 对象也带有一个 `operator[]` ，它返回Matrix的一个元素（例如，一个 `double` 的引用）。因此，他们使用类似 `m[i][j]` 的语法来访问矩阵的元素，而不是[象 `m(i,j)` 的语法。

数组的数组方案显然可以工作，但相对于 `operator()` 方法来说，缺乏灵活性。尤其是，用 `[][]` 方法很难表现的时候，用 `operator()` 方法可以很简单的完成，因此 `[][]` 方法很可能导致差劲的表现，至少某些情况细是这样的。

例如，实现 `[][]` 方法的最简单途径就是使用作为密集矩阵的，以以行为主的形式保存（或以列为主，我记不清了）的物理布局。相反， `operator()` 方法完全隐藏了矩阵的物理布局，在这种情况下，它可能带来更好的表现。

可以这么认为： `operator()` 方法永远不比 `[][]` 方法差，有时更好。

- `operator()` 永远不差，是因为用 `operator()` 方法实现以行为主的密集矩阵的物理布局非常容易。因此，当从性能观点出发，那样的结构正好是最佳布局时， `operator()` 方法也和 `[][]` 方法一样简单（也许 `operator()` 方法更容易一点点，但我不想夸大其词）。
- `operator()` 方法有时更好，是因为当对于给定的应用，有其它比以行为主的密集矩阵更好的布局时，用 `operator()` 方法比 `[][]` 方法实现会容易得多。

作为一个物理布局使得实现困难的例子，最近的项目发生在以列访问矩阵元素（也就是，算法访问一列中的所有元素，然后是另一列等），如果物理布局是以行为主的，对矩阵的访问可能会“cache失效”。例如，如果行的大小几乎和处理器的cache大小相当，那么对每个元素的访问，都会发生“cache不命中”。在这个特殊的项目中，我们通过将映射从逻辑布局（行，列）变为物理布局（列，行），性能得到了20%的提升。

当然，还有很多这类事情的例子，而稀疏矩阵在这个问题中则是又一类例子。通常，使用 `operator()` 方法实现一个稀疏矩阵或交换行／列顺序更容易， `operator()` 方法不会损失什么，而可能获得一些东西——它不会更差，却可能更好。

使用 `operator()` 方法。

13.10 该从外（接口优先）还是从内（数据优先）设计类？

从外部！

良好的接口提供了一个简化的，以用户词汇表达的视图。在面向对象软件的情况下，接口通常是单个类或一组紧密结合的类的public方法的集合.

首先考虑对象的逻辑特征是什么，而不是打算如何创建它。例如，假设要创建一个 `Stack` （栈）类，其包含一个 `LinkedList` ：

```
class Stack {
public:
  _// ..._
private:
  LinkedList list_;
};
```

Stack是否应该有一个返回 `LinkedList` 的 `get()` 方法？或者一个带有 `LinkedList` 的 `set()` 方法？或者一个带有 `LinkedList` 的构造函数？显然，答案是“不”，因为应该从外向里设计接口。也就是说， `Stack` 对象的用户并不关心 `LinkedList` ；他们只关心 pushing 和 popping。

现在看另一个更微妙的例子。假设 `LinkedList` 类使用 `Node` 对象的链表来创建，每一个 `Node` 对象有一个指向下一个 `Node` 的指针：

```
class Node { /*...*/ };

class LinkedList {
public:
  // ...
private:
  Node* first_;
};
```

`LinkedList` 类是否应该有一个让用户访问第一个 `Node` 的 `get()` 方法？ `Node` 对象是否应该有一个让用户访问链中下一个 `Node` 的 `get()` 方法？换句话说，从外部看， `LinkedList` 应该是什么样的？ `LinkedList` 是否实际上就是一个 `Node` 对象的链？或者这些只是实现的细节？如果只是实现的细节， `LinkedList` 将如何让用户在某时刻访问 `LinkedList` 中的每一个元素？

某人的回答： `LinkedList` 不是的 `Node` 链。它可能的确是用 `Node` 创建的，但这不是本质。它的本质是元素的序列。因此， `LinkedList` 抽象应该提供一个“LinkedListIterator”，并且“LinkedListIterator”应该有一个 `operator++` 来访问下一个元素，并且有一对 `get()` / `set()` 来访问存储于 `Node` 的值（ `Node` 元素中的值只由 `LinkedList` 用户负责，因此有一对 `get()` / `set()` 以允许用户自由地维护该值）。

从用户的观点出发，我们可能希望 `LinkedList` 类支持看上去类似使用指针算法访问数组的运算符：

```
void userCode(LinkedList& a)
{
  for (LinkedListIterator p = a.begin(); p != a.end(); ++p)
    std::cout << *p << '\n';
}
```

实现这个接口， `LinkedList` 需要一个 `begin()` 方法和 `end()` 方法。它们返回一个“LinkedListIterator”对象。该“LinkedListIterator”需要一个前进的方法， `++p` ；访问当前元素的方法， `*p` ；和一个比较运算符， `p != a.end()` 。

如下的代码，关键在于 `LinkedList` 类没有任何让用户访问 `Node` 的方法。`Node` 作为实现技术被完全地隐藏了。 `LinkedList` 类内部可能用双重链表取代，甚至是一个数组，区别仅仅在于一些诸如`prepend(elem)` 和 `append(elem)`方法的性能上。

```
#include <cassert>    // Poor man's exception handling

class LinkedListIterator;
class LinkedList;

class Node {
  // No public members; this is a "private class"_
  friend LinkedListIterator;   // 友员类
  friend LinkedList;
  Node* next_;
  int elem_;
};

class LinkedListIterator {
public:
  bool operator== (LinkedListIterator i) const;
  bool operator!= (LinkedListIterator i) const;
  void operator++ ();   // Go to the next element
  int& operator*  ();   // Access the current element
private:
  LinkedListIterator(Node* p);
  Node* p_;
  friend LinkedList;  // so LinkedList can construct a LinkedListIterator
};

class LinkedList {
public:
  void append(int elem);    // Adds elem after the end_
  void prepend(int elem);   // Adds elem before the beginning
  // ...
  LinkedListIterator begin();
  LinkedListIterator end();
  // ...
private:
  Node* first_;
};
```

这些是显然可以内联的方法（可能在同一个头文件中）：

```
inline bool LinkedListIterator::operator== (LinkedListIterator i) const
{
  return p_ == i.p_;
}

inline bool LinkedListIterator::operator!= (LinkedListIterator i) const
{
  return p_ != i.p_;
}

inline void LinkedListIterator::operator++()
{
  assert(p_ != NULL);  // or if (p_==NULL) throw ...
  p_ = p_->next_;
}

inline int& LinkedListIterator::operator*()
{
  assert(p_ != NULL);  // or if (p_==NULL) throw ...
  return p_->elem_;
}

inline LinkedListIterator::LinkedListIterator(Node* p)
  : p_(p)
{ }

inline LinkedListIterator LinkedList::begin()
{
  return first_;
}

inline LinkedListIterator LinkedList::end()
{
  return NULL;
}
```

结论：链表有两种不同的数据。存储于链表中的元素的值由链表的用户负责（并且只有用户负责，链表本身不阻止用户将第三个元素变成第五个），而链表底层结构的数据（如 `next` 指针等）值由链表负责（并且只有链表负责，也就是说链表不让用户改变（甚至看到！）可变的 `next` 指针）。

因此 `get()` / `set()` 方法只获取和设置链表的元素，而不是链表的底层结构。由于链表隐藏了底层的指针等结构，因此它能够作非常严格的承诺（例如，如果它是双重链表，它可以保证每一个后向指针都被下一个 `Node` 的前向指针匹配）。

我们看了这个例子，类的一些数据的值由用户负责（这种情况下需要有针对数据的 `get()` / `set()` 方法），但对于类所控制的数据则不必有 `get()` / `set()` 方法。

注意：这个例子的目的不是为了告诉你如何写一个链表类。实际上不要自己做链表类，而应该使用编译器所提供的“容器类”的一种。理论上来说，要使用标准容器类之一，如：`std::list<T>` 模板。

[14] 友元

FAQs in section [14]:

14.1 什么是友元（`friend`）？

允许另一个类或函数访问你的类的东西。

友元可以是函数或者是其他的类。类授予它的友元特别的访问权。通常同一个开发者会出于技术和非技术的原因，控制类的友元和成员函数（否则当你想更新你的类时，还要征得其它部分的拥有者的同意）。

14.2 友元破坏了封装吗？

如果被适当的使用，实际上可以增强封装。

当一个类的两部分会有不同数量的实例或者不同的生命周期时，你经常需要将一个类分割成两部分。在这些情况下，两部分通常需要直接存取彼此的数据（这两部分原来在同一个类中，所以你不必增加直接存取一个数据结构的代码；你只要将代码改为两个类就行了）。实现这种情况的最安全途径就是使这两部分成为彼此的友元。

如果你象刚才所描述的那样使用友元，就可以使私有的（`private`）保持私有。不理解这些的人在以上这种情形下还天真的想避免使用友元，他们要么使用公有的（`public`）数据（罕见！），要么通过公有的 `get()` 和 `set()` 成员函数使两部分可以访问数据。而他们实际上破坏了封装。只有当在类外（从用户的角度）看待私有数据仍“有意义”时，为私有数据设置公有的 `get()` 和 `set()` 成员函数才是合理的。在许多情况下，这些 `get()` / `set()` 成员函数和公有数据一样差劲：它们仅仅隐藏了私有数据的名称，而没有隐藏私有数据本身。

同样，如果你将友元函数当做一种类的 `public:` 存取函数的语法不同的变种来使用的话，友元函数就和破坏封装的成员函数一样会破坏封装。换一种说法，类的友元不会破坏封装的壁垒：和类的成员函数一样，它们就是封装的壁垒。

14.3 使用友元函数的优缺点是什么？

友元函数在接口设计选择上提供了一定程度的自由。

成员函数和友元函数具有同等的特权（100% 的）。主要的不同在于友元函数象`f(x)`这样调用，而成员函数象`x.f()`这样调用。因此，可以在成员函数（`x.f()`）和友元函数（`f(x)`）之间选择的能力允许设计者选择他所认为更具可读性的语法来降低维护成本。

友元函数主要缺点是需要额外的代码来支持动态绑定时。要得到虚友元（`virtual` `friend`）的效果，友元函数应该调用一个隐藏的（通常是`protected:`）虚。例如：

```
class Base {
public:
  friend void f(Base& b);
  // ...
protected:
  virtual void do_f();
  // ...
};

inline void f(Base& b)
{
  b.do_f();
}

class Derived : public Base {
public:
  // ...
protected:
  virtual void do_f();  // "覆盖" f(Base& b)的行为
  // ...
};

void userCode(Base& b)
{
  f(b);
}
```

在`userCode(Base&)`中的`f(b)`语句将调用虚拟的`b.do_f()`。这意味着如果`b`实际是一个派生类的对象，那么`Derived::do_f()`将获得控制权。注意派生类覆盖的是保护的虚（`protected:` `virtual`）成员函数`do_f()`；而不是它友元函数`f(Base&)`。

14.4 “友元关系既不继承，也不传递”是什么意思？

仅仅因为我承认对你的友情，允许你访问我，并不自动地允许你的孩子访问我，并不自动地允许你的朋友访问我，并不自动地允许我访问你。

- 我不见得信任我朋友的孩子。友元的特权不被继承。友元的派生类不一定是友元。如果`Fred`类声明`Base`类是友元，那么`Base`类的派生类不会自动地被赋予对于`Fred`的对象的访问特权。
- 我不见得信任我朋友的朋友。友元的特权不被传递。友元的友元不一定是友元。如果`Fred`类声明`Wilma`类是友元，并且`Wilma`类声明`Betty`类是友元，那么`Betty`类不

会自动地被赋予对于 `Fred` 的对象的访问特权。
- 你不见得仅仅因为我声称你是我的朋友就信任我。友元的特权不是自反的。如果 `Fred` 类声明 `Wilma` 类是友元，则 `Wilma` 对象拥有访问 `Fred` 对象的特权，但 `Fred` 对象不会自动地拥有对 `Wilma` 对象的访问特权。

14.5 类应该使用成员函数还是友元函数？

尽量使用成员函数，不得已时使用友元。

有时在语法上，友元更好（例如，`Fred` 类中，友元函数允许 `Fred` 参数作为第二个参数，而成员函数必须是第一个）。另一个好的用法是二元中缀运算符。例如，如果你想允许 `aFloat + aComplex` 的话，`aComplex + aComplex` 应该被定义为友元而不是成员函数。（成员函数不允许提升左边的参数，因为那样会改变成员函数调用对象的类）。

在其他情况下，首选成员函数。

[15] 通过 `<iostream>` 和 `<cstdio>` 输入／输出

FAQs in section [15]:

15.1 为什么应该 `<iostream>` 而不是传统的 `<cstdio>` ？

因为`<iostream>`加强了类型安全，减少了错误，提升了性能，可扩展，并且提供继承。

`printf()` 不错，`scanf()` 不管其可能导致错误，也还是有价值的，然而对于 C++ I／O（译注：I／O即输入／输出） 所能做的来说，它们的功能都是非常有限的。相对于C （使用 `printf()` 和 `scanf()`）来说，C++ I／O （使用 `<<` 和 `>>` ）是：

- 更好的类型安全：使用 `<iostream>` ，编译器静态地知道被 I／O 的对象的类型。相反，`<cstdio>`使用“`%`”域来动态地指出类型。
- 更少的错误倾向：使用 `<iostream>`，没有多余的必须与实际被 I／O的对象相一致的“`%`”。去除多余的，意味着去除了一类错误。
- _可扩展：_C++ `<iostream>` 机制允许在不破坏现有代码的情况下，新的用户定义类型能够被I／O。（可以想象一下，每个人同事添加新的不相容的“`%`”域到`printf()` 和 `scanf()`，是怎样的混乱场面？！）。
- _可继承：_C++ `<iostream>` 机制是建立在真正的类上的，如 `std::ostream` 和 `std::istream` 。不象 `<cstdio>` 的 `FILE*` ，有真正的类，因此可继承。这意味着你可以

你可以拥有其他的用户定义的看上去以及其效果都类似流的东西，而它可以做任何你需要的奇怪的和有趣的事情。你将自动的得到无数行的你所不认识的用户写的 I／O代码，并且，他们不需要认识你写的“extended stream”类。

15.2 当键入非法字符时，为何我的程序进入死循环？

举个例子，假设你有如下的代码，从`std::cin`读取一个整数：

```
#include <iostream>

int main()
{
  std::cout << "Enter numbers separated by whitespace (use -1 to quit): ";
  int i = 0;
  while (i != -1) {
    std::cin >> i;        // 不良的形式 — 见如下注释
    std::cout << "You entered " << i << '\n';
  }
}
```

该程序没有检查键入的是否是合法字符。尤其是，如果某人键入的不是整数（如“x”），`std::cin`流进入“失败状态”，并且其后所有的输入尝试都不作任何事情而立即返回。换句话说，程序进入了死循环；如果`42`是最后成功读到的数字，程序会反复打印“`You entered 42`”消息。

检查合法输入的一个简单方法是将输入请求从`while`循环体中移到`while`循环的控制表达式，如：

```
#include <iostream>

int main()
{
  std::cout << "Enter a number, or -1 to quit: ";
  int i = 0;
  while (std::cin >> i) {    // 良好的形式
    if (i == -1) break;
    std::cout << "You entered " << i << '\n';
  }
}
```

这样的结果就是当你敲击end-of-file，或键入一个非整数，或键入`-1`时，while 循环会退出。

（自然，你也可以不用`break`，而将`while`循环表达式`while (std::cin >> i)`改为`((std::cin >> i) && (i != -1))`，但这不是本FAQ的重点，本 FAQ 处理iostream，而不是一般的结构化编程指南。）

15.3 那个古怪的 `while (std::cin >> foo)` 语法如何工作？

“古怪的 `while (std::cin >> foo)` 语法”的例子见前一个FAQ。

`(std::cin >> foo)` 表达式调用了适当的 `operator>>`（例如，它调用了左边带有 `std::istream` 参数以及，如果的类型是 `int`，并且右边有一个 `int&` 的 `operator>>`）。 `std::istream` `operator>>` 函数按惯例地返回左边的参数，在这里，它返回 `std::cin`。下一步编译器注意到返回的 `std::istream` 处于一个布尔型的上下文中，因此编译器将 `std::istream` 转换为一个布尔值。

编译器调用一个称为 `std::istream::operator void*()` 的成员函数来将 `std::istream` 转换成布尔。它返回一个被转换成布尔的 `void*` 指针（ `NULL` 成为 `false`，任何其他的指针成为 `true`）。因此在这里，编译器产生了 `std::cin.operator void*()` 的调用，就如同你象 `(void*) std::cin` 这样显式地强制类型转换。

如果stream处于良好状态，那么转换算符 `operator void*()` 返回非指针，如果处于失败状态，则返回 `NULL`。例如，如果读了太多次（也就是说，已经处于end-of-file），或实际输入到流的信息不是 `foo` 的合法类型（如，如果 `foo` 是一个 `int`，而数据是一个“x”字符），流会进入失败状态并且转换算符会返回 `NULL`。

`operator>>` 不是简单地返回一个 `bool`（或 `void*`）以支出是否成功或失败的原因是为了支持“级联”语法：

```
std::cin >> foo >> bar;
```

`operator>>` 是向左结合的，意味着如上的代码会解释为：

```
(std::cin >> foo) >> bar;
```

换句话说，如果我们将 `operator>>` 变为一个普通的函数名称，如 `readFrom()`，将变为这样的表达式：

```
readFrom( readFrom(std::cin, foo), bar);
```

我们总是从最内部开始计算表达式。因为 `operator>>` 的左结合性，就成了最左边表达式 `std::cin >> foo`。该表达式返回 `std::cin`（更合适的，他返回一个它左边参数的引用）给下一个表达式。下一个表达式也返回（一个引用）给 `std::cin`，但第二个引用被忽略了，因为它是这个“表达式语句”的最外边的表达式了。

15.4 为何我的输入处理会超过文件末尾？

因为只有在试图超过文件末尾后，eof标记才会被设置。也就是，在从文件读最后一个字节时，还没有设置 eof 标记。例如，假设输入流映射到键盘——在这种情况下，理论上来说，C++库不可能预知到用户所键入的字符是否是最后一个字符。

如，如下的代码对于计数器 `i` 会有“超出 1”的错误：

```
int i = 0;
while (! std::cin.eof()) {   // 错误！（不可靠）
  std::cin >> x;
  ++i;
  // Work with x ...
}
```

你实际需要的是：

```
int i = 0;
while (std::cin >> x) {       // 正确！（可靠）
  ++i;
  // Work with x ...
}
```

15.5 为什么我的程序在第一个循环后，会忽略输入请求呢？

因为数字的提取器将非数字留在了输入缓冲器之后。

如果你的代码看上去象这样：

```
char name[1000];
int age;

for (;;) {
  std::cout << "Name: ";
  std::cin >> name;
  std::cout << "Age: ";
  std::cin >> age;
}
```

而你实际需要的是：

```
for (;;) {
  std::cout << "Name: ";
  std::cin >> name;
  std::cout << "Age: ";
  std::cin >> age;
  std::cin.ignore(INT_MAX, '\n');
}
```

当然，你也许想将 `for (;;)` 语句变为 `while (std::cin)` ，但不要搞错，在循环末尾通过 `std::cin.ignore(...);` 这一行跳过非数字字符。

15.6 如何为 `class Fred` 提供打印？

用算符重载提供一个友元的左切换的算符 `operator<<` 。

```
#include <iostream>

class Fred {
public:
  friend std::ostream& operator<< (std::ostream& o, const Fred& fred);
  // ...
private:
  int i_;    // 只是为了说明
};

std::ostream& operator<< (std::ostream& o, const Fred& fred)
{
  return o << fred.i_;
}

int main()
{
  Fred f;
  std::cout << "My Fred object: " << f << "\n";
}
```

由于 `Fred` 对象是 `<<` 算符的右边的操作数，我们使用非成员函数（在这里是一个友元）。如果 `Fred` 对象被期望为在 `<<` 的左边（那就是 `myFred << std::cout` 而不是 `std::cout << myFred` ），则就会有一个命名为 `operator<<` 的成员函数。

注意， `operator<<` 返回流。这就使得输出算符能够被级联。

15.7 但我可以总是使用 `printOn()` 方法而不是一个友元函数吗？

不。

通常人们总是愿意使用 `printOn()` 方法而不是一个友元函数的原因是因为他们错误地相信友元破坏了封装并且／或者友元是不良的。这些信仰是天眞的和错误的：适当的使用，友元实际上可以增强封装。

这也不是说 `printOn()` 方法没用。例如：为一个完整的继承层次的类提供打印时就是有用的。但如果你看到一个 `printOn()` 方法，它通常应该是 `protected` 的，而不是 `public` 的。

为完整，这里给出“ `printOn()` 方法”。想法是有一个成员函数（通常被称为 `printOn()` ，来完成实际的打印，然后有一个 `operator<<` 来调用 `rintOn()` 方法）。当错误地完成它时， `printOn()` 方法是 `public` 的，因此 `operator<<` 不需要成为友元——它成为一个简单的顶级函数，即不是类的友元，也不是类的成员函数。这是一些示例代码：

```
#include <iostream>

class Fred {
public:
  void printOn(std::ostream& o) const;
  // ...
};

// operator<< 可以被声明为非友元 [不推荐！]
std::ostream& operator<< (std::ostream& o, const Fred& fred);

// 实际打印由内部的 printOn() 方法完成 [不推荐！]
void Fred::printOn(std::ostream& o) const
{
  // ...
}

// operator<< 调用 printOn() [不推荐！]
std::ostream& operator<< (std::ostream& o, const Fred& fred)
{
  fred.printOn(o);
  return o;
}
```

人们错误地假定“由于避免了出现一个友元函数”而减少了维护成本。这个假定是错误的，因为：

1. 在维护成本上，“顶级函数调用成员”方法不会带来任何好处。我们假设 *N* 行代码来完成实际的打印。在使用友元函数的情况下，那 *N* 行代码将直接访问类的 `private` / `protected` 部分，这意味着某人无论何时改变了类的 `private` / `protected` 部分，那 *N* 行代码将需要被扫描并且可能被修改，这增加了维护成本。然而，使用 `printOn()` 方法并没有改变：我们仍然有 *N* 行代码直接访问类的 `private` / `protected` 部分。因此将代码从友元函数移到成员函数根本就并不减少维护成本。没有减少。在维护成本上没有好处。（如果有的话，`printOn()` 方法更差一点，因为你有了一个额外的原先没有的函数，现在有更多行的代码需要被维护）
2. “顶级函数调用成员”方法使得类更难被使用，尤其是程序员不是类的设计者时。这种方法将一个并不期望被调用的 `public` 方法暴露给程序员。当程序员阅读类的 `public` 方法时，他们会看见两种方法做同一件事情。文档需要象这样说明：“这个和那个并不完全一样，但不要用这个；而应该用那个”。并且通常的程序员会说：“唔？如果我不应该使用它，为什么它是 `public` 的？”事实上 `printOn()` 方法是 `public` 的唯一理由是避免将友元授权给 `operator<<` ，这个主张对于某些仅仅想使用这个类的程序员来说，是微妙的并且难以理解的。

总之，“顶级函数调用成员”方法有成本，没有收益。因此，通常，不是好主意。

注意：如果 `printOn()` 方法是 `protected` 或 `private` 的，第二个异议将不成立。有些情况这方法是合理的，如为一个完整的继承层次的类提供打印时。同样要注意，当 `printOn()` 方法是非 `public` 的时， `operator<<` 需要成为友元。

15.8 如何为 `class Fred` 提供输入？

使用算符重载](operator-overloading.html)提供一个友元的右切换的算符`operator>>`。除了参数没有一个`const`："`Fred&`"而不是"`const Fred&`"，其他和[输出算符类似。

```
#include <iostream>

class Fred {
public:
  friend std::istream& operator>> (std::istream& i, Fred& fred);
  // ...
private:
  int i_;    // 只是为了说明
};

std::istream& operator>> (std::istream& i, Fred& fred)
{
  return i >> fred.i_;
}

int main()
{
  Fred f;
  std::cout << "Enter a Fred object: ";
  std::cin >> f;
  // ...
}
```

注意`operator>>`返回流。这就使得输入算符能被级联和／或在循环或语句中使用。

15.9 如何为完整继承层次的类提供打印？

提供一个友元调用一个`protected` `virtual`函数：

```
class Base {
public:
  friend std::ostream& operator<< (std::ostream& o, const Base& b);
  // ...
protected:
  virtual void printOn(std::ostream& o) const;
};

inline std::ostream& operator<< (std::ostream& o, const Base& b)
{
  b.printOn(o);
  return o;
}

class Derived : public Base {
protected:
  virtual void printOn(std::ostream& o) const;
};
```

最终结果是`operator<<`就象是动态绑定，即使它是一个友元函数。这被称为"虚友元函数用法"。

注意派生类重写了`printOn(std::ostream&)` `const`。尤其是，它们不提供他们自己的`operator<<`。

自然的，如果 `Base` 是一个ABC（抽象基类），`Base::printOn(std::ostream&) const` 可以用“ `= 0` ”语法被声明为纯虚函数。

15.10 在DOS 和／或 OS/2环境下，如何以二进制模式“重打开” `std::cin` 和 `std::cout` ？

这依赖于实现，请查看你的编译器的文档。

例如，假设你想使用 `std::cin` 和 `std::cout` 进行二进制 I／O。更假设你的操作系统（如 DOS 或 OS/2）坚持将从 `std::cin` 输入的“ `\r\n` ”翻译为“ `\n` ”，将从 `std::cout` 或 `std::cerr` 输出的“ `\n` ”翻译为“ `\r\n` ”。

不行的是没有标准方法使得 `std::cin` ，`std::cout` 和／或 `std::cerr` 以二进制模式被打开。关闭流并且试图以二进制方式重打开它们，可能会得到非期望的或不合需要的结果。

在系统的区别处，实现可能提供了一种方法使它们成为二进制流，但你必须查看手册来找到。

15.11 为何我不能在如“ `..\test.dat` ”这样的不同的目录打开文件？

因为“ `\t` ”是一个 tab 字符。

你应该在文件中使用正斜杠，即使在使用反斜杠的操作系统，如 DOS, Windows, OS/2等中。例如：

```
#include <iostream>
#include <fstream>

int main()
{
  #if 1
    std::ifstream file("../test.dat");  _// 正确！_
  #else
    std::ifstream file("..\test.dat");  _// 错误！_
  #endif

  _// ..._
}
```

记住，反斜杠（“ `\` ”）被用来在字符串中建立特殊字符：“ `\n` ”是换行，“ `\b` ”是退格，以及“ `\t` ”是一个tab，“ `\a` ”是一个警告（alert），“ `\v` ”是一个vertical-tab等。因此文件名“ `\version\next\alpha\beta\test.dat` ”被解释为一堆有区的字符；应该用“ `/version/next/alpha/beta/test.dat` ”来替代，即使系统中使用“ `\` ”作为目录分隔符，如 DOS, Windows, OS/2等。这是因为操作系统中的库例程是可交换地处理“/”和“\”的。

15.12 如何将一个值（如，一个数字）转换为 `std::string`？

有两种方法：可以使用 `<stdio>` 工具或 `<iostream>` 库。通常，你应该使用 `<iostream>` 库。

`<iostream>` 库允许你使用如下的语法（转换一个 `double` 的示例，但你可以替换美妙的多的任何使用 `<<` 算符的东西）将任何美妙得多的东西转换为 `std::string`：

```
#include <iostream>
#include <sstream>
#include <string>

std::string convertToString(double x)
{
  std::ostringstream o;
  if (o << x)
    return o.str();
  // 这儿进行一些错误处理...
  return "conversion error";
}
```

`std::ostringstream` 对象 `o` 提供了类似 `std::cout` 提供的格式化工具。你可以使用操纵器和格式化标志来控制格式化的结果，就如同你用 `std::cout` 可以做到的。

在这个例子中，我们通过被重载了的插入运算符 `<<`，将 `x` 插入到 `o`。它调用了iostream的格式化工具将 `x` 转换为一个 `std::string`。`if` 测试保证转换正确工作——对于内建／固有类型，总是成功的，但 `if` 测试是良好的风格。

表达式 `os.str()` 返回包含了被插入到流 `o` 中的任何东西的 `std::string`，在这里，是 x 的值的字符串。

15.13 如何将 `std::string` 转换为数值？

有两种方法：可以使用 `<stdio>` 工具或 `<iostream>` 库。通常，你应该使用 `<iostream>` 库。

`<iostream>` 库允许你使用如下的语法（转换一个 `double` 的示例，但你可以替换美妙的多的任何能使用 `>>` 算符被读取的东西）将一个 `std::string` 转换为美妙得多的任何东西：

```
#include <iostream>
#include <sstream>
#include <string>

double convertFromString(const std::string& s)
{
  std::istringstream i(s);
  double x;
  if (i >> x)
    return x;
  // 这儿进行一些错误处理...
  return 0.0;
}
```

`std::istringstream` 对象 `i` 提供了类似`std::cin`提供的格式化工具。你可以使用操纵器和格式化标志来控制格式化的结果，就如同你用`std::cin`能做到的。

在这个示例中，我们传递了`td::string` `s`来初始化`std::istringstream` `i` （例如，`s` 可能是字符串“`123.456`”），然后我们通过被重载了的抽取运算符 `>>`，将 `i` 抽取到 `x`。它调用了iostream的格式化工具对字符串进行尽可能的／适当的基于 x 的类型的转换。

`if` 测试保证了转换正确地工作。例如，如果字符串包含不适合 x 类型的字符，`if` 测试将失败。

[16] 自由存储（Freestore）管理

FAQs in section [16]:

16.1 `delete p` 删除指针 `p` ，还是删除指针所指向的数据 `*p` ？

指针指向的数据。

关键字应该是 `delete_the_thing_pointed_to_by` 。同样的情况也发生在 C中释放指针所指的内存： `free(p)` 实际上是指 `free_the_stuff_pointed_to_by(p)` 。

16.2 可以 `free()` 一个由 `new` 分配的指针吗？可以 `delete` 一个由 `malloc()` 分配的指针吗？

不！

在一个程序中同时使用 `malloc()` 和 `delete` 或者同时使用 `new` 和 `free()` 是合情合理合法的。但是，对由 `new` 分配的指针调用 `free()` ，或对由 `malloc()` 分配的指针调用 `delete` ，是无理的、非法的、卑劣的。

当心！我偶尔收到一些人的e-mail，他们告诉我在他们的机器 X 上和编译器 Y 上工作正常。但这并不能使得它成为正确的！有时他们说：“但我只是用一下字符数组而已”。即使虽然如此，也不要在同一个指针上混合 `malloc()` 和 `delete` ，或在同一个指针上混合 `new` 和 `free()` 。如果通过 `p = new char[n]` 分配，则必须使用 `delete[] p` ；不可以使用 `free(p)` 。如果通过分配 `p = malloc(n)` ，则必须使用 `free(p)` ；不可以使用 `delete[] p` 或 `delete p` ！将它们混合，如果将代码放到新的机器上，新的编译器上，或只是同样编译器的新版本上，都可能导致运行时灾难性的失败。

记住这个警告。

16.3 为什么要用 `new` 取代原来的值得信赖的 `malloc()` ？

构造函数／析构函数，类型安全，可覆盖性（Overridability）。

- 构造函数／析构函数：与 `malloc(sizeof(Fred))` 不一样， `new Fred()` 调用 `Fred` 的构造函数。同样， `delete p` 调用 `*p` 的析构函数。
- 类型安全： `malloc()` 返回一个没有类型安全的 `void*` 。 `new Fred()` 返回一个正确类型（一个 `Fred*` ）的指针。
- 可覆盖性： `new` 是一个可被类重写／覆盖的算符（ `operator` ），而 `malloc()` 在类上没有可覆盖性。

16.4 可以在一个由 `new` 分配的指针上使用 `realloc()` 吗？

不可！

`realloc()` 拷贝时，使用的是位拷贝（*bitwise* copy ）算符，这会打碎许多 C++ 对象。C++对象应该被允许拷贝它们自己。它们使用自己的拷贝构造函数或者赋值算符。

除此之外，`new` 使用的堆可能和 `malloc()` 和 `realloc()` 使用的堆不同！

16.5 需要在 `p = new Fred()` 之后检查 `NULL` 吗？

不！（但如果你只有旧的编译器，你可能不得不强制 new 算符在内存溢出时抛出一个异常。）

总是在每一个 `new` 调用之后写显式的 `NULL` 测试实在是非常痛苦的.如下的代码是非常单调乏味的：

```
Fred* p = new Fred();
if (p == NULL)
  throw std::bad_alloc();
```

如果你的编译器不支持（或如果你拒绝使用）异常， 你的代码可能会更单调乏味：

```
Fred* p = new Fred();
if (p == NULL) {
  std::cerr << "Couldn't allocate memory for a Fred" << endl;
  abort();
}
```

振作一下。在 C++中，如果运行时系统无法为 `p = new Fred()` 分配 `sizeof(Fred)` 字节的内存，会抛出一个 `std::bad_alloc` 异常。与 `malloc()` 不同， `new` 永远不会返回 `NULL` ！

因此你只要简单地写：

```
Fred* p = new Fred();   // 不需要检查 `p` 是否为 `NULL`
```

然而，如果你的编译器很古老，它可能还不支持这个。查阅你的编译器的文档找到“ `new` ”。如果你只有古老的编译器，就必须强制编译器拥有这种行为。

16.6 我如何确信我的（古老的）编译器会自动检查 `new` 是否返回 `NULL` ？

最终你的编译器会支持的。

如果你只有古老的不自动执行 `NULL` 测试的编译器的话，你可以安装一个“new handler”函数来强制运行时系统来测试。你的“new handler”函数可以作任何你想做的事情，诸如抛出一个异常， `delete` 一些对象并返回（在 `operator new` 会试图再分配的情况下），打印一个消息或者从程序中 `abort()` 等等。

这里有一个“new handler”的例子，它打印消息并抛出一个异常。它使用 `std::set_new_handler()` 被安装：

```
#include <new>       // 得到 std::set_new_handler
#include <cstdlib>   // 得到 abort()
#include <iostream>  // 得到 std::cerr

class alloc_error : public std::exception {
public:
  alloc_error() : exception() { }
};

void myNewHandler()
{
  // 这是你自己的 handler。它可以做任何你想要做的事情。
  throw alloc_error();
}

int main()
{
  std::set_new_handler(myNewHandler);   // 安装你的 "new handler"
  // ...
}
```

在 `std::set_new_handler()` 被执行后，如果／当内存不足时，`operator new` 将调用你的 `myNewHandler()`。这意味着 `new` 不会返回 `NULL`：

```
Fred* p = new Fred();   // 不需要检查 `p` 是否为 `NULL`
```

注意：如果你的编译器不支持异常处理，作为最后的诉求，你可以将 `throw ...;` 这一行改为：

```
std::cerr << "Attempt to allocate memory failed!" << std::endl;
abort();
```

注意：如果某些全局的／静态的对象的构造函数使用了 `new`，由于它们的构造函数在 `main()` 开始之前被调用，因此它不会使用 `myNewHandler()` 函数。不幸的是，没有简便的方法确保 `std::set_new_handler()` 在第一次使用 `new` 之前被调用。例如，即使你将 `std::set_new_handler()` 的调用放在全局对象的构造函数中，你仍然无法知道包含该全局对象的模块（“编译单元”）被首先还是最后还是还是中间某个位置被解释。因此，你仍然无法保证 `std::set_new_handler()` 的调用会在任何其他全局对象的构造函数调用之前。

16.7 在 `delete p` 之前需要检查 `NULL` 吗？

不需要！

C++语言担保，如果 `p` 等于 `NULL`，则 `delete p` 不作任何事情。由于之后可以得到测试，并且大多数的测试方法论都强制显式测试每个分支点，因此你不应该加上多余的 `if` 测试。

错误的：

```
if (p != NULL)
  delete p;
```

正确的：

```
delete p;
```

16.8 `delete p` 执行了哪两个步骤？

`delete p` 是一个两步的过程：调用析构函数，然后释放内存。`delete p` 产生的代码看上去是这样的（假设是 `Fred*` 类型的）：

```
// 原始码：delete p;
if (p != NULL) {
  p->~Fred();
  operator delete(p);
}
```

`p->~Fred()` 语句调用 `p` 指向的 `Fred` 对象的析构函数。

`operator delete(p)` 语句调用内存释放原语 `void operator delete(void* p)` 。该原语类似 `free(void* p)` 。（然而注意，它们两个不能互换；举例来说，没有谁担保这两个内存释放原语会使用同一个堆！）。

16.9 在 `p = new Fred()` 中，如果 `Fred` 构造函数抛出异常，是否会内存“泄漏”？

不会。

如果异常发生在 `p = new Fred()` 的 `Fred` 构造函数中， C++语言确保已分配的 `sizeof(Fred)` 字节的内存会自动从堆中回收。

这里有两个细节： `new Fred()` 是一个两步的过程：

1. `sizeof(Fred)` 字节的内存使用 `void* operator new(size_t nbytes)` 原语被分配。该原语类似于 `malloc(size_t nbytes)` 。（然而注意，他们两个不能互换；举例来说，没有谁担保这两个内存分配原语会使用同一个堆！）。
2. 它通过调用 `Fred` 构造函数在内存中建立对象。第一步返回的指针被作为 `this` 参数传递给构造函数。这一步被包裹在一个块中以处理这步中抛出异常的情况。

因此实际产生的代码可能是象这样的：

```
// 原始代码：Fred* p = new Fred();
Fred* p = (Fred*) operator new(sizeof(Fred));
try {
  new(p) Fred();       // Placement new
} catch (...) {
  operator delete(p);  // 释放内存_
  throw;               // 重新抛出异常
}
```

标记为“Placement `new` ”的这句语句调用了 `Fred` 构造函数。指针 `p` 成了构占函数 `Fred::Fred()` 内部的 `this` 指针。

16.10 如何分配／释放一个对象的数组？

使用 `p = new T[n]` 和 `delete[] p`：

```
Fred* p = new Fred[100];
// ...
delete[] p;
```

任何时候你通过 `new` 来分配一个对象的数组（通常在表达式中有 `[ n ]` ），则在 `delete` 语句中必须使用 `[]` 。该语法是必须的，因为没有什么语法可以区分指向一个对象的指针和指向一个对象数组的指针（从 C 派生出的某些东西）。

16.11 如果 `delete` 一个由 `new T[n]` 分配的数组，漏了 `[]` 会如何？

所有生命毁灭性地终止。

正确地连接 `new T[n]` 和 `delete[] p` 是程序员的——不是编译器的——责任。如果你弄错了，编译器会在编译时或运行时给出错误消息。堆（Heap）被破坏是可能的结果，或者更糟糕，你的程序可能会死亡。

16.12 当 `delete` 一个内建类型 (`char` , `int` , 等)的数组时，能去掉 `[]` 吗？

不行！

有时程序员会认为在 `delete[] p` 中存在 `[]` 仅仅是为了编译器为数组中的每个元素调用适当的析构函数。由于这个原因，他们认为一些内建类型的数组，如 `char` 或 `int` 可以不需要 `[]` 。举例来说，他们认为以下是合法的代码：

```
void userCode(int n)
{
  char* p = new char[n];
  // ...
  delete p;     // <— 错！应该是 delete[] p ！
}
```

但以上代码是错误的，并且会导致一个运行时的灾难。更详细地来说，`delete p`调用的是`operator delete(void*)`，而`delete[] p`调用的是`operator delete[](void*)`。虽然后者的默认行为是调用前者，但将后者用不同的行为取代是被允许的（这种情况下通常也会将相应的`operator new[](size_t)`中的 `new` 取代）。如果被取代的`delete[]` 代码与`delete` 代码不兼容，并且调用错误的那个（例如，你写了`delete p`而不是`delete[] p`），在运行时可能完蛋。

16.13 `p = new Fred[n]`之后，编译器在`delete[] p`的时候如何知道有个对象被析构？

精简的回答：魔法。

详细的回答：运行时系统将对象的数量 `n` 保存在某个通过指针 `p` 可以获取的地方。有两种普遍的技术来实现。这些技术都在商业编译器中使用，各有权衡，都不完美。这些技术是：

- 超额分配数组并将 `n` 放在第一个`Fred`对象的左边。
- 使用关联数组， `p` 作为键， `n` 作为值。

16.14 成员函数调用`delete this`合法吗？

只要你小心，一个对象请求自杀(`delete` `this`).是可以的。

以下是我对“小心”的定义：

1. 你必须100%的确定，`this`对象是用 `new`分配的（不是用`new]`，也不是用[定位放置`new`，也不是一个栈上的局部对象，也不是全局的，也不是另一个对象的成员，而是明白的普通的`new`）。
2. 你必须100%的确定，该成员函数是`this`对象最后调用的的成员函数。
3. 你必须100%的确定，剩下的成员函数（`delete` `this`之后的）不接触到 `this`对象任何一块（包括调用任何其他成员函数或访问任何数据成员）。
4. 你必须 100%的确定，在`delete` `this`之后不再去访问`this`指针。换句话说，你不能去检查它，将它和其他指针比较，和 `NULL`比较，打印它，转换它，对它做任何事。

自然，对于这种情况还要习惯性地告诫：当你的指针是一个指向基类类型的指针，而没有虚析构函数时（也不可以`delete` `this`）。

16.15 如何用 `new` 分配多维数组？

有许多方法，取决于你想要让数组有多大的灵活性。一个极端是，如果你在编译时就知道数组的所有的维数，则可以静态地（就如同在C中）分配多维数组：

```
class Fred { /*...*/ };
void someFunction(Fred& fred);

void manipulateArray()
{
  const unsigned nrows = 10;  // 行数是编译期常量
  const unsigned ncols = 20;  // 列数是编译期常量
  Fred matrix[nrows][ncols];

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // 访问(i,j)元素的方法：
      someFunction( matrix[i][j] );

      // 可以安全地“返回”，不需要特别的delete代码：
      if (today == "Tuesday" && moon.isFull())
        return;     // 月圆的星期二赶紧退出
    }
  }

  // 在函数末尾也没有显式的delete代码
}
```

更一般的，矩阵的大小只有到运行时才知道，但确定它是一个矩形。这种情况下，你需要使用堆（“自由存储”）（heap，freestore），但至少你可以把所有元素非胚在自由存储块中。

```
void manipulateArray(unsigned nrows, unsigned ncols)
{
  Fred* matrix = new Fred[nrows * ncols];

  // 由于我们上面使用了简单的指针，因此我们需要非常
  // 小心避免漏过 delete 代码。
  // 这就是为什么要捕获所有异常：
  try {

    // 访问(i,j) 元素的方法：
    for (unsigned i = 0; i < nrows; ++i) {
      for (unsigned j = 0; j < ncols; ++j) {
        someFunction( matrix[i*ncols + j] );
      }
    }

    // 如果你想在月圆的星期二早点退出，
    // 就要确保在返回的所有途径上做 delete ：
    if (today == "Tuesday" && moon.isFull()) {
      delete[] matrix;
      return;
    }

    // ...

  }
  catch (...) {
    // 确保在异常抛出后delete ：
    delete[] matrix;
    throw;    // 重新抛出当前异常
  }

  // 确保在函数末尾也做了 delete ：
  delete[] matrix;
}
```

最后是另一个极端，你可能甚至不确定矩阵是矩形的。例如，如果每行可以有不同的长度，你就需要为个别地分配每一行。在如下的函数中，`ncols[i]` 是第 `i` 行的列数，`i` 的可变范围是 `0` 到 `nrows-1` 。

```
void manipulateArray(unsigned nrows, unsigned ncols[])
{
  typedef Fred* FredPtr;

  // 如果后面抛出异常，不要成为漏洞：
  FredPtr* matrix = new FredPtr[nrows];

  // 以防万一稍后会有异常，将每个元素设置为 NULL：
  // (见 try 块顶端的注释。)
  for (unsigned i = 0; i < nrows; ++i)
    matrix[i] = NULL;

  // 由于我们上面使用了简单的指针，我们需要
  // 非常小心地避免漏过delete 代码。
  // 这就是为什么我们要捕获所有的异常：
  try {

    // 接着我们组装数组。如果其中之一抛出异常，所有的
    // 已分配的元素都会被释放 (见如下的 catch )。
    for (unsigned i = 0; i < nrows; ++i)
      matrix[i] = new Fred[ ncols[i] ];

    // 访问(i,j) 元素的方法：
    for (unsigned i = 0; i < nrows; ++i) {
      for (unsigned j = 0; j < ncols[i]; ++j) {
        someFunction( matrix[i][j] );
      }
    }

    // 如果你想在月圆的星期二早些退出，
    // 确保在返回的所有途径上做 delete：
    if (today == "Tuesday" && moon.isFull()) {
      for (unsigned i = nrows; i > 0; --i)
        delete[] matrix[i-1];
      delete[] matrix;
      return;
    }

    // ...

  }
  catch (...) {
    // 确保当有异常抛出时做 delete  :
    // 注意 matrix[...] 中的一些指针可能是
    // NULL, 但由于delete NULL是合法的，所以没问题。
    for (unsigned i = nrows; i > 0; --i)
      delete[] matrix[i-1];
    delete[] matrix;
    throw;    // 重新抛出当前异常
  }

  // 确保在函数末尾也做 delete  :
  // 注意释放与分配反向：
  for (unsigned i = nrows; i > 0; --i)
    delete[] matrix[i-1];
  delete[] matrix;
}
```

注意释放过程中 `matrix[i-1]` 的使用。这样可以防止无符号值 `i` 的步进为小于0 的回绕。

最后，注意指针和数组是会带来麻烦的](containers-and-templates.html#[31.1])。通常，最好将你的指针封装在一个有着安全的和简单的接口的类中。[下一个FAQ告诉你如何这样做。

16.16 但前一个FAQ的代码太技巧容易出错！有更简单的方法吗？

有。

前一个FAQ之所以太过技巧而容易出错是因为它使用了指针，我们知道指针和数组会带来麻烦](containers-and-templates.html#[31.1])。解决办法是将指针封装到一个有着安全的和简单的接口的类中。例如，我们可以定义一个 `Matrix` 类来处理矩形的矩阵，用户代码将比[前一个FAQ中的矩形矩阵的代码简单得多：

```
// Matrix 类的代码在下面显示...
void someFunction(Fred& fred);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix matrix(nrows, ncols);   // 构造一个 matrix

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      _// 访问(i,j) 元素的方法：_
      someFunction( matrix(i,j) );

      _// 你可以不用写任何的 delete 代码安全地“返回”：
      if (today == "Tuesday" && moon.isFull())
        return;     // 月圆的星期二早些退出
    }
  }

  // 在函数末尾也没有显式的delete代码
}
```

需要注意的主要是整理后的代码的短小。例如，再如上的代码中没有任何 `delete` 语句，也不会有内存泄漏，这个假设仅仅是基于析构函数正确地完成它的工作。

以下就是使得以上成为可能的 `Matrix` 的代码：

```
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // 如果任何一个尺寸为 0，则抛出 BadSize 对象的异常：
  class BadSize { };

  // 基于大三法则（译注：即三者须同时存在）：
 ~Matrix();
  Matrix(const Matrix& m);
  Matrix& operator= (const Matrix& m);

  // 取得 (i,j) 元素的访问方法：
  Fred&       operator() (unsigned i, unsigned j);
  const Fred& operator() (unsigned i, unsigned j) const;
  // 如果i 或j 太大，抛出BoundsViolation 对象
  class BoundsViolation { };

private:
  Fred* data_;
  unsigned nrows_, ncols_;
};

inline Fred& Matrix::operator() (unsigned row, unsigned col)
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

inline const Fred& Matrix::operator() (unsigned row, unsigned col) const
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

Matrix::Matrix(unsigned nrows, unsigned ncols)
  : data_  (new Fred[nrows * ncols]),
    nrows_ (nrows),
    ncols_ (ncols)
{
  if (nrows == 0 || ncols == 0)
    throw BadSize();
}

Matrix::~Matrix()
{
  delete[] data_;
}
```

注意以上的 `Matrix` 类完成两件事：将技巧性的内存管理代码从客户代码（例如， `main()` ）移到类中，并且总体上减少了编程。这第二点很重要。例如，假设 `Matrix` 有略微的可重用性，将复杂性从 `Matrix` 的用户们〔复数〕处移到了 `Matrix` 自身〔单数〕就等于将复杂性从多的方面移到少的方面。任何看过星际旅行2的人都知道多数的利益高于少数或者个体的利益。

16.17 但上面的 `Matrix` 类是针对 `Fred` 的！有办法使它通用吗？

有；那就是使用模板：

以下就是如何能用模板：

```
#include "Fred.hpp"     // 得到Fred类的定义
// Matrix<T> 的代码在后面显示...
void someFunction(Fred& fred);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix<Fred> matrix(nrows, ncols);   // 构造一个称为matrix的 Matrix<Fred>

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // 访问 (i,j) 元素的方法：
      someFunction( matrix(i,j) );

      // 你可以不用任何的delete 的代码安全地“返回”：
      if (today == "Tuesday" && moon.isFull())
        return;     // 月圆的星期二早些退出
    }
  }

  // 函数末尾也没有显式的delete代码

}
```

现在很容易为非 `Fred` 的类使用 `Matrix<T>`。例如，以下为`std::string` 使用一个 `Matrix`（ `std::string` 是标准字符串类）：

```
#include <string>

void someFunction(std::string& s);

void manipulateArray(unsigned nrows, unsigned ncols)
{
  Matrix<std::string> matrix(nrows, ncols);   // 构造一个 Matrix<std::string>

  for (unsigned i = 0; i < nrows; ++i) {
    for (unsigned j = 0; j < ncols; ++j) {
      // 访问 (i,j) 元素的方法：
      someFunction( matrix(i,j) );

      // 你可以不用任何的delete 的代码安全地“返回”：
      if (today == "Tuesday" && moon.isFull())
        return;     // 月圆的星期二早些退出
    }
  }

  // 函数末尾也没有显式的delete代码
}
```

因此，你可以从模板得到类的完整家族。例如， `Matrix<Fred>`, `Matrix<std::string>`, `Matrix< Matrix<std::string> >` 等等。

以下是实现该模板的一种方法：

```
template<class T>  // 详见模板一节
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // 如果任何一个尺寸为 0，则抛出 BadSize 对象
  class BadSize { };

  // 基于大三法则（译注：即三者须同时存在）：
 ~Matrix();
  Matrix(const Matrix<T>& m);
  Matrix<T>& operator= (const Matrix<T>& m);

  // 获取 (i,j) 元素的访问方法：
  T&       operator() (unsigned i, unsigned j);
  const T& operator() (unsigned i, unsigned j) const;
  // 如果 i 或 j 太大，则抛出 BoundsViolation 对象
  class BoundsViolation { };

private:
  T* data_;
  unsigned nrows_, ncols_;
};

template<class T>
inline T& Matrix<T>::operator() (unsigned row, unsigned col)
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

template<class T>
inline const T& Matrix<T>::operator() (unsigned row, unsigned col) const
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row*ncols_ + col];
}

template<class T>
inline Matrix<T>::Matrix(unsigned nrows, unsigned ncols)
  : data_  (new T[nrows * ncols])
  , nrows_ (nrows)
  , ncols_ (ncols)
{
  if (nrows == 0 || ncols == 0)
    throw BadSize();
}

template<class T>
inline Matrix<T>::~Matrix()
{
  delete[] data_;
}
```

16.18 还有其他方法建立 `Matrix` 模板吗？

用标准的 `vector` 模板，制作一个向量的向量。

以下代码使用了一个 `vector<vector<T> >`（注意两个 `>` 符号之间的空格）。

```
#include <vector>

template<class T>  // 详见模板一节
class Matrix {
public:
  Matrix(unsigned nrows, unsigned ncols);
  // 如果任何的尺寸为 0，抛出 BadSize 对象
  class BadSize { };

  // 不需要大三法则！
  // 得到 (i,j) 元素的访问方法：
  T&       operator() (unsigned i, unsigned j);
  const T& operator() (unsigned i, unsigned j) const;
  // 如果 i 或 j 太大，则抛出 BoundsViolation 对象
  class BoundsViolation { };

private:
  vector<vector<T> > data_;
};

template<class T>
inline T& Matrix<T>::operator() (unsigned row, unsigned col)
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row][col];
}

template<class T>
inline const T& Matrix<T>::operator() (unsigned row, unsigned col) const
{
  if (row >= nrows_ || col >= ncols_) throw BoundsViolation();
  return data_[row][col];
}

template<class T>
Matrix<T>::Matrix(unsigned nrows, unsigned ncols)
  : data_ (nrows)
{
  if (nrows == 0 || ncols == 0)
    throw BadSize();
  for (unsigned i = 0; i < nrows; ++i)
    data_[i].resize(ncols);
}
```

16.19 C++ 有能够在运行期指定长度的数组吗？

有，是基于标准库有一个 `std::vector` 模板可以提供这种行为的认识。

没有，是基于内建数组类型需要在编译期指定其长度的认识。

有，是基于即使对于内建数组类型也可以在运行期指定第一维索引边界的认识。例如，看一下前一个FAQ，如果你只需要数组的第一维的维数具有灵活性，你可以申请一个新的数组的数组，而不是一个指向多个数组的指针数组：

```
const unsigned ncols = 100;           // ncols = 数组的列数

class Fred { /*...*/ };

void manipulateArray(unsigned nrows)  // nrows = 数组的行数
{
  Fred (*matrix)[ncols] = new Fred[nrows][ncols];
  // ...
  delete[] matrix;
}
```

如果你所需要的不是在运行期改变数组的第一维维数，则不能这么做。

但非万不得已，不要用数组。因为数组是会带来麻烦的。如果可以的话，使用某些类的对象。万不得已才用数组。

16.20 如何使类的对象总是通过 `new` 来创建而不是局部的或者全局的／静态的对象？

使用命名的构造函数用法。

就如命名的构造函数用法的通常做法，所有构造函数是`private:` 或`protected:` ，且有一个或多个`public` `static` `create()`方法（因此称为“命名的构造函数，named constructors”），每个构造函数对应一个。此时， `create()` 方法通过 `new` 来分配对象。由于构造函数本身都不是`public` ，因此没有其他方法来创建该类的对象。

```
class Fred {
public:
  // create() 方法就是 "命名的构造函数，named constructors":
  static Fred* create()                 { return new Fred();     }
  static Fred* create(int i)            { return new Fred(i);    }
  static Fred* create(const Fred& fred) { return new Fred(fred); }
  // ...

private:
  // 构造函数本身是 private 或 protected:
  Fred();
  Fred(int i);
  Fred(const Fred& fred);
  // ...
};
```

这样，创建 `Fred` 对象的唯一方法就是通过 `Fred::create()` ：

```
int main()
{
  Fred* p = Fred::create(5);
  // ...
  delete p;
}
```

如果你希望 `Fred`有派生类，则须确认构造函数在 `protected:` 节中。

注意，如果你想允许 `Fred` 类的对象成为 `Wilma` 类的成员，可以把 `Wilma` 作为 `Fred` 的友元。当然，这样会软化最初的目标，也就是强迫 `Fred` 对象总是通过 `new` 来分配。

16.21 如何进行简单的引用计数？

如果你所需要的只是分发指向同一个对象的多个指针，并且当最后一个指针消失的时候能自动释放该对象的能力的话，你可以使用类似如下的“只能指针（smart pointer）”类：

```
// Fred.h

class FredPtr;

class Fred {
public:
  Fred() : count_(0) /*...*/ { }  // 所有的构造函数都要设置 count to 0 !
  // ...
private:
  friend FredPtr;     // 友元类
  unsigned count_;
  // count_ 必须被所有构造函数初始化
  // count_ 就是指向 this的对FredPtr象数目
};

class FredPtr {
public:
  Fred* operator-> () { return p_; }
  Fred& operator* ()  { return *p_; }
  FredPtr(Fred* p)    : p_(p) { ++p_->count_; }  // p 不能为 NULL
 ~FredPtr()           { if (--p_->count_ == 0) delete p_; }
  FredPtr(const FredPtr& p) : p_(p.p_) { ++p_->count_; }
  FredPtr& operator= (const FredPtr& p)
        { // 不要改变这些语句的顺序！
          // (如此的顺序适当的处理了自赋值)
          ++p.p_->count_;
          if (--p_->count_ == 0) delete p_;
          p_ = p.p_;
          return *this;
        }
private:
  Fred* p_;    // p_ 永远不为 NULL
};
```

自然，你可以使用嵌套类，将 `FredPtr` 改名为 `Fred::Ptr` 。

注意，在构造函数，拷贝构造函数，赋值算符和析构函数中增加一点检查，就可以软化上面的“不远不为 NULL”的规则。如果你这样做的话，可能倒不如在“ `*` ”和“ `->` ”算符中放入一个 `p_ != NULL` 检查（至少是一个 `assert()` ）。我不推荐 `operator Fred*()` ，因为它可能让人们意外地取得 `Fred*` 。

`FredPtr` 的隐含约束之一是它可能指向通过 `new` 分配的 `Fred` 对象。如果要真正的安全，可以使所有的 `Fred` 构造函数成为 `private` ，为每个构造函数加一个用 `new` 来分配 `Fred` 对象且返回一个 `FredPtr` （不是 `Fred*` ）的 `public` (`static`) `create()` 方法来加强这个约束。这种办法是创建 `Fred` 对象而得到一个 `FredPtr` 的唯一办法（“ `Fred* p = new Fred()` ”会被“ `FredPtr p = Fred::create()` ”取代）。这样就没人会意外破坏引用计数的机制了。

例如，如果 `Fred` 有一个 `Fred::Fred()` 和一个 `Fred::Fred(int i, int j)` ， `class` `Fred` 会变成：

```
class Fred {
public:
  static FredPtr create();              // 定义如下的 class FredPtr {...}
  static FredPtr create(int i, int j);  // 定义如下的 class FredPtr {...}
  // ...
private:
  Fred();
  Fred(int i, int j);
  // ...
};

class FredPtr { /* ... */ };

inline FredPtr Fred::create()             { return new Fred(); }
inline FredPtr Fred::create(int i, int j) { return new Fred(i,j); }
```

最终结果是你现在有了一种办法来使用简单的引用计数为给出的对象提供“指针语义（pointer semantics）”。 `Fred` 类的用户明确地使用 `FredPtr` 对象，它或多或少的类似 `Fred*` 指针。这样做的好处是用户可以建立多个 `FredPtr` “智能指针”对象的拷贝，当最后一个 `FredPtr` 对象消失时，它所指向的 `Fred` 对象会被自动释放。

如果你希望给用户以“引用语义”而不是“指针语义”的话，可以使用引用计数提供“写时拷贝（copy on write）”。

16.22 如何用写时拷贝（copy-on-write）语义提供引用计数？

引用计数可以由指针语义或引用语义完成。前一个FAQ显示了如何使用指针语义进行引用计数。本FAQ将显示如何使用引用语义进行引用计数。

基本思想是允许用户认为他们在复制 `Fred` 对象，但实际上真正的实现并不进行复制，直到一些用户试图修改隐含的 `Fred` 对象才进行真正的复制。

`Fred::Data` 类装载了 `Fred` 类所有的数据。 `Fred::Data` 也有一个额外的成员 `count_` ，来管理引用计数。 `Fred` 类最后成了一个指向 `Fred::Data` 的“智能指针”（内部的）。

```
class Fred {
public:

  Fred();                               // 默认构造函数
  Fred(int i, int j);                   // 普通的构在函数

  Fred(const Fred& f);
  Fred& operator= (const Fred& f);
 ~Fred();

  void sampleInspectorMethod() const;   // this 对象不会变
  void sampleMutatorMethod();           // 会改变 this o对象
  // ...

private:
```

```
  class Data {
  public:
    Data();
    Data(int i, int j);
    Data(const Data& d);

    // 由于只有 Fred 能访问 Fred::Data 对象，
    // 只要你愿意，你可以使得 Fred::Data的数据为 public，
    // 但如果那样使你不爽，就把数据作为 private
    // 还要用friend Fred;使 Fred 成为友元类
    // ...

    unsigned count_;
    // count_ 是指向的this的Fred 对象的数目
    // count_ m必须被所有的构造函数初始化为 1
    // (从 1 开始是因为它被创建它的Fred 对象所指)
  };

  Data* data_;
};

Fred::Data::Data()              : count_(1) /*初始化其他数据*/ { }
Fred::Data::Data(int i, int j)  : count_(1) /*初始化其他数据*/ { }
Fred::Data::Data(const Data& d) : count_(1) /*初始化其他数据*/ { }

Fred::Fred()             : data_(new Data()) { }
Fred::Fred(int i, int j) : data_(new Data(i, j)) { }

Fred::Fred(const Fred& f)
  : data_(f.data_)
{
  ++ data_->count_;
}

Fred& Fred::operator= (const Fred& f)
{
  // 不要更该这些语句的顺序！
  // (如此的顺序适当地处理了自赋值)
  ++ f.data_->count_;
  if (--data_->count_ == 0) delete data_;
  data_ = f.data_;
  return *this;
}

Fred::~Fred()
{
  if (--data_->count_ == 0) delete data_;
}

void Fred::sampleInspectorMethod() const
{
  // 该方法承诺 (“const”) 不改变 *data_中的任何东西
  // 除此以外，任何数据访问将简单地使用“data_->...”
}

void Fred::sampleMutatorMethod()
{
  // 该方法可能需要改变 *data_中的数据
  // 因此首先检查this是否唯一的指向 *data_
  if (data_->count_ > 1) {
    Data* d = new Data(*data_);    // 调用 Fred::Data的拷贝构造函数
    -- data_->count_;
    data_ = d;
  }
  assert(data_->count_ == 1);

  // 现在该方法如常进行“data_->...”的访问
}
```

如果非常经常地调用 `Fred` 的默认构造函数，你可以为所有通过`Fred::Fred()`构造的`Fred`共享一个公共的`Fred::Data` 对象来消除那些 `new` 调用。为避免静态初始化顺序问题，该共享的 `Fred::Data` 对象在一个函数内“首次使用”时才创建。如下就是对以上的代码做的改变（注意，该共享的`Fred::Data`对象的析构函数永远不会被调用；如果这成问题的话，要么解决静态初始化顺序的问题，要么索性返回到如上描述的方法）：

```
class Fred {
public:
  // ...
private:
  // ...
  static Data* defaultData();
};

Fred::Fred()
: data_(defaultData())
{
  ++ data_->count_;
}

Fred::Data* Fred::defaultData()
{
  static Data* p = NULL;
  if (p == NULL) {
    p = new Data();
    ++ p->count_;    // 确保它不会成为 0
  }
  return p;
}
```

注意：如果 `Fred` 通常作为基类的话，也可以为类层次提供引用计数。

16.23 如何为派生类提供写时拷贝（copy-on-write）语义的引用计数？

前一个FAQ给出了引用语义的引用计数策略，但迄今为止都针对单个类而不是分层次的类。本FAQ扩展之前的技术以允许为类层次提供引用计数。基本不同之处在于现在`Fred::Data`是类层次的根，着可能使得它有一些虚函数。注意 `Fred` 类本身仍然没有任何的虚函数。

虚构造函数用法用来建立 `Fred::Data` 对象的拷贝。要选择创建哪个派生类，如下的示例代码使用了命名构造函数用法，但还有其它技术（构造函数中加一个`switch`语句等）。示例代码假设了两个派生类：`Der1`和`Der2`。派生类的方法并不查觉引用计数。

```
class Fred {
public:

  static Fred create1(const std::string& s, int i);
  static Fred create2(float x, float y);

  Fred(const Fred& f);
  Fred& operator= (const Fred& f);
 ~Fred();

  void sampleInspectorMethod() const;   // this 对象不会被改变
  void sampleMutatorMethod();           // 会改变 this 对象
```

```
  // ...

private:

  class Data {
  public:
    Data() : count_(1) { }
    Data(const Data& d) : count_(1) { }              // 不要拷贝 'count_' 成员！
    Data& operator= (const Data&) { return *this; }  // 不要拷贝 'count_' 成员！
    virtual ~Data() { assert(count_ == 0); }         // 虚析构函数
    virtual Data* clone() const = 0;                 // 虚构造函数
    virtual void sampleInspectorMethod() const = 0;  // 纯虚函数
    virtual void sampleMutatorMethod() = 0;
  private:
    unsigned count_;   // count_ 不需要是 protected 的
    friend Fred;       // 允许Fred 访问 count_
  };

  class Der1 : public Data {
  public:
    Der1(const std::string& s, int i);
    virtual void sampleInspectorMethod() const;
    virtual void sampleMutatorMethod();
    virtual Data* clone() const;
    // ...
  };

  class Der2 : public Data {
  public:
    Der2(float x, float y);
    virtual void sampleInspectorMethod() const;
    virtual void sampleMutatorMethod();
    virtual Data* clone() const;
    // ...
  };

  Fred(Data* data);
  // 创建一个拥有 *data 的 Fred 智能引用
  // 它是 private 的以迫使用户使用 createXXX() 方法
  // 要求：data 必能为 NULL

  Data* data_;   // Invariant: data_ is never NULL
};

Fred::Fred(Data* data) : data_(data)  { assert(data != NULL); }

Fred Fred::create1(const std::string& s, int i) { return Fred(new Der1(s, i)); }
Fred Fred::create2(float x, float y)            { return Fred(new Der2(x, y)); }

Fred::Data* Fred::Der1::clone() const { return new Der1(*this); }
Fred::Data* Fred::Der2::clone() const { return new Der2(*this); }

Fred::Fred(const Fred& f)
  : data_(f.data_)
{
  ++ data_->count_;
}

Fred& Fred::operator= (const Fred& f)
{
  // 不要更该这些语句的顺序！
  // (如此的顺序适当地处理了自赋值)
  ++ f.data_->count_;
  if (--data_->count_ == 0) delete data_;
  data_ = f.data_;
  return *this;
}

Fred::~Fred()
{
  if (--data_->count_ == 0) delete data_;
}
```

```
void Fred::sampleInspectorMethod() const
{
  // 该方法承诺 ("const") 不改变*data_中的任何东西
  // 因此我们只要"直接把方法传递"给 *data_：
  data_->sampleInspectorMethod();
}

void Fred::sampleMutatorMethod()
{
  // 该方法可能需要更该 *data_中的数据
  // 因此首先检查this 是否唯一的指向*data_
  if (data_->count_ > 1) {
    Data* d = data_->clone();   // 虚构造函数用法
    -- data_->count_;
    data_ = d;
  }
  assert(data_->count_ == 1);

  // 现在"直接把方法传递给" *data_：
  data_->sampleInspectorMethod();
}
```

自然，`Fred::Der1` 和 `Fred::Der2` 的构造函数和 `sampleXXX` 方法将需要被以某种途径适当的实现。

16.24 你能绝对地防止别人破坏引用计数机制吗？如果能的话，你会这么做吗？

不能，（通常）不会。

有两个基本的办法破坏引用计数机制：

1. 如果某人获得了 `Fred*` （而不是别强制使用的 `FredPtr` ），该策略就会被破坏。如果 `FredPtr` 类有返回一个 `Fred&` 的 `operator*()` 的话，就可能得到 `Fred*` ：`FredPtr p = Fred::create(); Fred* p2 = &*p;` 。是的，那是奇异的、不被预期的，但它可能发生。该漏洞有两个方法弥补：重载 `Fred::operator&()` 使它返回一个 `FredPtr` ，或改变 `FredPtr::operator*()` 的返回类型，使它返回一个 `FredRef` （ `FredRef` 是一个模拟引用的类；它需要拥有 `Fred` 所拥有的所有方法，并且需要将这些方法的调用转送给隐含的 `Fred` 对象；第二种选择可能成为性能瓶颈，这取决于编译器在内联方法中的表现）。另一个方法是消除 `FredPtr::operator*()` ——相应的会失去取得和使用 `Fred&` 的能力。但即使你这样做了，某些人仍然可以通过显式的调用 `operator->()` ：`FredPtr p = Fred::create(); Fred* p2 = p.operator->();` 来取得一个 `Fred*` 。
2. 如果某人有一个泄漏的和／或悬空的 `FredPtr` 指针的话，该策略会被破坏。基本上我们说 `Fred` 是安全的，但我们无法阻止别人对 `FredPtr` 对象做傻事。（并且如果我们可以通过 `FredPtrPtr` 对象来解决的话，则对于 `FredPtrPtr` 仍然有相同的问题）。这里的一个漏洞是如果某人使用 `new` 创建了一个 `FredPtr` ，然后 `FredPtr` 就可能有泄漏（这里最糟的情况是有泄漏，但通常还是比悬空指针要好一点点）。该漏洞可以通过将 `FredPtr::operator new()` 声明为 `private` 来弥补，从而防止 `new FredPtr()` 。此处另

一个漏洞是如果某人创建了一个局部的 `FredPtr` 对象，则可取得 `FredPtr` 的地址并传递给 `FredPtr*`。如果 `FredPtr*` 生存期比 `FredPtr` 更长，就可能成为悬空指针——颤抖的指针。该漏洞可以通过防止取得 `FredPtr` 的地址来弥补（重载 `FredPtr::operator&()` 为 `private`），相应的会损失一些功能。但即使你这样做了，他们只要这样做：`FredPtr p; ... FredPtr& q = p;`（或者将 `FredPtr&` 传递其它什么），仍然可以创建 `FredPtr*` 与一样危险的 `FredPtr&`。

并且，即使我们弥补了所有那些漏洞，C++ 还有奇妙的称为指针转换（pointer cast）的语法。使用一两个指针转换，一个有意的程序员可以创造一个大得足以穿过一辆卡车的漏洞。

此处的教训是：(a) 无论你多么的智者千虑，也不可能防止间谍，(b) 你可以简单的防止错误。

因此我建议：用易建易用的机制来防止错误，不要操心试图去防止间谍。即使你殚精竭力做了，也不会成功，得不偿失。

如果不能使用C++语言本身来防止间谍，还有其它办法吗？有。我为它亲自用旧式风格的代码检视。由于间谍技巧通常包括一些奇异的语法和／或指针转换的使用和联合（union），你可以使用工具来指出大多数的“是非之地”。

16.25 在C++中能使用垃圾收集吗？

能。

相比于前面所述的“智能指针”技术，垃圾收集技术：

- 更轻便
- 通常更有效 （尤其当平均的对象尺寸较小时或多线程环境中）
- 能处理数据中的“循环（cycles）”（如果数据结构能形成循环，引用计数技术通常会有“泄漏”）
- 有时会泄漏其它对象（由于垃圾收集器必要的保守性，有时会进入一个看上去象是指针的随机位模式的分配单元，尤其是如果分配单元较大时，可能导致该分配单元有泄漏）。
- 与现存的库工作得更好（由于智能指针需要显式使用，可能很难集成到现存的库中）

16.26 C++的两种垃圾收集器是什么？

通常，好像有两种风味的C++垃圾收集器：

1. 保守的垃圾收集器。这些垃圾收集器对于栈和C++对象的分布知之甚少或一无所知，只是寻找看上去象指针的位模式。实践中与 C 以及 C++ 代码共同工作，尤其是平均的对象尺寸较小时，这里有一些例子，按字母顺序：

- Boehm-Demers-Weiser collector
- Geodesic Systems collector

2. 混合的垃圾收集器。这些垃圾收集器通常适当地扫描栈，但需要程序员提供堆对象的布局信息。这需要程序员方面做更多工作，但结果是提高性能。这里有一些例子，按字母顺序：

    - Bartlett's mostly copying collector
    - Attardi and Flagella's CMM (如果谁有 URL，请发给我）。

由于C++垃圾收集器通常是保守的，如果一个位模式“看上去”象是有可能是指向另外一个未使用块的指针，就会有泄漏。当指向某块的指针实际超出了块（这是非法的，但一些程序员会越过该限制；唉）以及（很少）当一个指针被编译器的优化所隐藏，也会使它困惑。在实践中，这些问题通常不严重，然而倘若收集器有一些关于对象布局的提示的话，可能会改善这些情况。

16.27 还有哪里能得到更多的C++垃圾收集信息？

更多信息，详见垃圾收集 FAQ。

[17] 异常和错误处理

FAQs in section [17]:

17.1 `try` / `catch` / `throw` 通过哪些方法来改善软件质量？

通过排除使用 `if` 语句的一个理由。

代替 `try` / `catch` / `throw` 的通常做法是返回一个返回代码（有时称为错误代码），调用者通过诸如 `if` 的条件语句明确地测试。例如，`printf()`，`scanf()` 和 `malloc()` 就是这样工作的：调用者被假定为会测试返回值来判断函数是否成功。

尽管返回代码技术有时是最适当的错误处理技术，但会产生增加不必要的 `if` 语句这样的令人讨厌的效果。

- 质量降级：众所周知，条件语句可能包含的错误大约十倍于其他类型的语句。因此，在其他都相同时，如果你能从代码中消除条件语句，你会得到更健壮的代码。
- 推迟面市：由于条件语句是分支点，而它们关系到白盒法测试时的测试条件的个数，因此不必要的条件语句会增加测试的时间总量。如果你没有走过每个分支点，那么你的代码中就会有在测试中没有被执行过的指令，直到用户／客户发现它，那就糟了。
- 增加开发成本：不必要的控制流程的复杂性增加了寻找bug，修复bug，和测试的工作。

因此，相对于通过返回代码和 `if` 来报告错误，使用 `try` / `catch` / `throw` 所产生更少有bug，更低的开发成本和更快面市的代码。当然，如果你的团队没有任何使用 `try` / `catch` / `throw` 的经验，你也许想先在一个玩具性的项目上使用一下，以便确定你明白正在做的事情——在把武器拿上真枪实弹的前线前，总应该演练一下吧。

17.2 如何处理构造函数的失败?

抛出一个异常。

构造函数没有返回类型，所以返回错误代码是不可能的。因此抛出异常是标记构造函数失败的最好方法。

如果你没有或者不愿意使用异常，这里有一种方法。如果构造函数失败了，构造函数可以把对象带入一种“僵尸”状态。你可以通过设置一个内部状态位使对象就象死了一样，即使从技术上来说，它仍然活着。然后加入一个查询（“检察员”）成员函数，以便类的用户能够通过检查这个“僵尸位”来确定对象是真的活着还是已经成为僵尸（也就是一个“活着的死对象”）。你也许想有另一个成员函数来检查这个僵尸位，并且当对象并不是真正活着的时候，执行一个 no-op（或者是更令人讨厌的如 `abort()` ）。这样做真的不漂亮，但是如果你不能（或者不想）使用异常的话，这是最好的方法了。

17.3 如何处理析构函数的失败？

往log文件中写一个消息。或打电话给Tilda舅妈。但不要抛出异常！

以下是为什么（扣好你的安全带）：

C++的规则是你绝对不可以在另一个异常的被称为“栈展开(stack unwinding)”的过程中时，从析构函数抛出异常。举例来说，如果某人写了 `throw Foo()` ，栈会被展开，以至 `throw Foo()` 和 `} catch (Foo e) {` 之间的所有的栈页面被弹出。这被称为栈展开*(statck unwinding)*

在栈展开时，栈页面中的所有的局部对象会被析构。如果那些析构函数之一抛出异常（假定它抛出一个 `Bar` 对象），C++运行时系统会处于无法决断的境遇：应该忽略 `Bar` 并且在 `} catch (Foo e) {` 结束？应该忽略 `Foo` 并且寻找 `} catch (Bar e) {` ？没有好的答案——每个选择都会丢失信息。

因此C++语言担保，当处于这一点时，会调用 `terminate()` 来杀死进程。突然死亡。

防止这种情况的简单方法是不要从析构函数中抛出异常。但如果你真的要聪明一点，你可以说当处理另一个异常的过程中时，不要从析构函数抛出异常。但在第二种情况中，你处于困难的境地：析构函数本身既需要代码处理抛出异常，还需要处理一些“其他东西”，调用者没有当析构函数检测到错误时会发生什么的担保（可能抛出异常，也可能做一些“其他事情”）。因此完整的解决方案非常难写。因此索性就做一些“其他事情”。也就是，不要从析构函数中抛出异常。

当然，由于总有一些该规则无效的境况，这些话不应该被“引证”。但至少99%的情况下，这是一个好规则。

17.4 如果构造函数会抛出异常，我该怎样处理资源？

对象中的每个数据成员应该清理自己。

如果构造函数抛出异常，对象的析构函数将不会运行。如果你的对象需要撤销一些已经做了的动作（如分配了内存，打开了一个文件，或者锁定了某个信号量），这些需要被撤销的动作必须被对象内部的一个数据成员记住。

例如，应该将分配的内存赋给对象的一个“智能指针”成员对象 `Fred` ，而不是分配内存给未被初始化的 `Fred*` 数据成员。这样当该智能指针消亡时，智能指针的析构函数将会删除 `Fred` 对象。标准类 `auto_ptr` 就是这种“智能指针”类的一个例子。你也可以写你自己的引用计数智能指针。你也可以用智能指针来指向磁盘记录或者其它机器上的对象。

17.5 当别人抛出异常时，我如何改变字符数组的字符串长度来防止内存泄漏？

如果你要做的确实需要字符串，那么不要使用 `char` 数组，因为数组会带来麻烦。应该用一些类似字符串类的对象来代替。

例如，假设你要得到一个字符串的拷贝，随意修改这个拷贝，然后在修改过的拷贝的字符串末尾添加其它的字符串。字符数组方法将是这样：

```
void userCode(const char* s1, const char* s2)
{
  // 制作s1的拷贝：
  char* copy = new char[strlen(s1) + 1];
  strcpy(copy, s1);

  // 现在我们有了一个指向分配了的自由存储的内存的指针，
  // w我们需要用一个try块来防止内存泄漏：
  try {

    // ... n现在我们随意乱动这份拷贝...
    // 将s2 添加到被修改过的 copy 末尾：
    // ... [在此处重分配 copy] ...
    char* copy2 = new char[strlen(copy) + strlen(s2) + 1];
    strcpy(copy2, copy);
    strcpy(copy2 + strlen(copy), s2);
    delete[] copy;
    copy = copy2;

    // ... 最后我们再次随意乱动拷贝...

  } catch (...) {
    delete[] copy;   // 得到一个异常时，防止内存泄漏
    throw;           // 重新抛出当前的异常
  }

  delete[] copy;     // 没有得到异常时，防止内存泄漏
}
```

象这样使用 `char*` s是单调的并且容易发生错误。为什么不使用一个字符串类的对象呢？你的编译器也许提供了一个字符串类，而且它可能比你自己写的 `char*` s更快，当然也更简单、更安全。例如，如果你使用了标准化委员会的字符串类 `std::string` ，你的代码看上去就会象这样：

```
#include <string>            // 让编译器找到 std::string 类

void userCode(const std::string& s1, const std::string& s2)
{
  std::string copy = s1;     // 制作s1的拷贝
  // ... 现在我们随意乱动这份拷贝...

  copy += s2;                // A将 s2 添加到被修改过的拷贝末尾
  // ... 最后我们再次随意乱动拷贝...
}
```

函数体中总共只有两行代码，而前一个例子中有12行代码。节省来自内存管理，但也有一些是来自于我们不必先式的调用`str_xxx_()`例程。这里有一些重点：

- 由于`std::string`自动处理了内存管理，当增长字符串时，我们不需要先式地写任何分配内存的代码。
- 由于`std::string`自动处理了内存管理，在结束时不需要 `delete[]` 任何东西。
- 由于`std::string`自动处理了内存管理，在第二个例子中不需要 `try` 块，即使某人会在某处抛出异常。

[18] const正确性

FAQs in section [18]:

18.1 什么是“`const`正确性”？

这是个好东西。意思是用 `const` 关键字来阻止 `const` 对象被修改。

例如，如果你要编写一个函数 `f()` ，它接收 `std::string` 类型的参数，并且想要对调用者保证不会修改调用者传过来的 `std::string` 参数，可以按以下方法声明 `f()`

- `void f1(const std::string& s); //传const引用`
- `void f2(const std::string* sptr); //传const指针`
- `void f3(std::string s); //传值`

在传 *`const`* 引用和传 *`const`* 指针的情况下，任何试图在 `f()` 内部修改 `std::string` 的行为都会在编译时被编译器标记为错误。这完全是在编译时做的，所以使用 `const` 没有运行时的空间或速度损失。在传值时（ `f3()` ），被调用函数获得了调用者 `std::string` 的一份拷贝。也就是说， `f3()` 可以修改这个拷贝，但返回时这个拷贝会被销毁。尤其是 `f3()` 无法修改调用者的 `std::string` 对象。

举个反例，如果想要编写一个函数`g()`，也是接收`std::string`，但想要告知调用者g()有可能会修改调用者的`std::string`对象。这时，可以按以下方法声明`g()`：

- `void g1(std::string& s); //传非const引用`
- `void g2(std::string* sptr); //传非const指针`

在这些函数中省去`const`，就是告诉编译器允许（但不强制）它们修改调用者的`std::string`对象。因此，这些`g()`函数可以把它们的`std::string`传递给任何`f()`函数，但只有`f3()`（通过传值接收参数）能够将其参数传递给`g1()`或`g2()`。如果`f1()`或`f2()`需要调用`g()`函数，必须给`g()`传递一份`std::string`的本地拷贝。`f1()`或`f2()`的参数不能直接传递给`g()`函数。例如：

```
void g1(std::string& s);

void f1(const std::string& s)
{
  g1(s);          // 编译错误，因为s是const的

  std::string localCopy = s;
  g1(localCopy);  // 正确，因为localCopy不是const的
}
```

当然，在上面的例子中，任何`g1()`所做的修改都会反映到`f1()`函数内的`localCopy`对象。特别是，通过`const`引用传递给`f1()`的参数不会被修改。

18.2 “`const`正确性”是如何与普通的类型安全有何联系？

将参数声明为`const`正是另外一种形式的类型安全。这就好像`const std::string`是与`std::string`不同的类一样。因为`const`变量缺少一些非`const`变量所具有的一些变更性操作（例如，可以想象以下，`const std::string`没有赋值操作符）。

如果你发现普通的类型安全有助于构建正确的系统（的确有帮助，尤其是对于大型系统来说），你会发现`const`正确性也有帮助。

18.3 我应该“尽早”还是“推迟”确定`const`正确性？

应该在最最最开始。

事后保证`const`正确性会导致一种滚雪球效应：每次你在一个地方添加了`const`会需要在四个更多的地方也添加`const`。

18.4 “`const Fred* p`”是什么意思？

意思是 `p` 是一个指向 `Fred` 类的指针，但不能通过 `p` 来修改 `Fred` 对象（当然 `p` 也可以是 `NULL` 指针）。

例如，假设 `Fred` 类有一个叫做 `inspect()` 的 `const` 成员函数，那么写 `p->inspect()` 是可以的。但如果 `Fred` 类有一个非 `const` 成员函数 `mutate()` ，那么写 `p->mutate()` 就是个错误（编译器会捕获这种错误；不会在运行时测试；因此 `const` 不会降低运行速度）。

18.5 “ `const Fred* p` ”、“ `Fred* const p` ”和“ `const Fr` ”有什么不同？

应该从右往左读指针声明。

- `const Fred* p` 表明 `p` 指向一个 `const` 的 `Fred` 对象—— `Fred` 对象不能通过 `p` 修改。
- `Fred* const p` 表明 `p` 是一个指向 `Fred` 对象的 `const` 指针——可以通过 `p` 修改 `Fred` 对象，但不能修改 `p` 本身。
- `cosnt Fred* const p` 表明“ `p` 是一个指向 `const Fred` 对象的 `const` 指针”——不能修改 `p` ，也不能通过 `p` 修改 `Fred` 对象。

18.6 “ `const Fred& x` ”是什么意思？

意思是 `x` 是 `Fred` 对象的一个别名，但不能通过 `x` 来修改 `Fred` 对象。

例如，假设 `Fred` 类有一个叫做 `inspect()` 的 `const` 成员函数，那么写 `x.inspect()` 是可以的。但如果 `Fred` 类有一个非 `const` 成员函数 `mutate()` ，那么写 `x.mutate()` 就是个错误（编译器会捕获这种错误；不会在运行时检查；因此 `const` 不会降低运行速度）。

18.7 “ `Fred& const x` ”有意义吗？

没意义。

为了理解这个声明，需要从右往左读这个声明。因此“ `Fred& const x` ”的意思是“ `x` 是一个指向 `Fred` 的 `const` 引用”。但这是多余的，因为引用本来就是 `const` 的。你不能重新绑定一个引用。不管有没有 `const` ，都不行。

换句话说，“ `Fred& cosnt x` ”在功能上与“ `Fred& x` ”是一样的。因为在 `&` 后面加上 `const` 没什么用，因此为了避免迷惑就不应该多此一举。有人可能会认为这里加上 `const` 后指向的 `Fred` 对象就是 `const` 的了，就好像“ `const Fred& x` ”一样。

18.8 “ `Fred const& x` ”是什么意思？

“`Fred cosnt& x`”在功能上与`const Fred& x`相同。然而，真正的问题是应该用哪一种。

答案：绝没有任何人能够为你所在的机构做决定，除非他们了解你的机构。没有放之四海而皆准的规则。没有对所有机构都“正确”的答案。所以不要让任何人做仓促的选择。“思考（Think）”并非一个四字母的单词。

例如，一些机构看重的是一致性，并且已经有大量代码使用“`const Fred&`”了。对于他们来说，不管是否有优点，“`Fred const&`”都不是个好选择。还有很多其它的商业环境，一些倾向于“`Fred const&`”，其它则倾向于“`const Fred&`”。

采用适合你机构中普通维护程序员的写法。不是专家，不是傻瓜，而是维护代码的普通程序员。除非你决定解雇他们并雇佣新人，否则就要确保他们能够理解你的代码。根据实际情况做商业决定，而不是根据其它什么人的假设。

使用“`Fred const&`”需要克服一些惯性。现在大多数C++书籍都使用`const Fred&`，大多数程序员学C++时接触的就是这种语法，并且仍然这么用。这并非是说`const Fred&`一定对你的机构好。但在更改（这种风格）期间，和/或在招收新人时，的确可能会引起一些混乱。一些机构认为用`Fred const&`带来的好处更大，其它机构则不这么认为。

另一个警告：如果决定用`Fred const&`，确保采取措施使人们不会误写成没意义的“`Fred& const x`”。

18.9 “`Fred const* x`”是什么意思？

“`Fred cosnt* x`”在功能上与`const Fred* x`相同。然而，真正的问题是应该用哪一种。

答案：绝没有任何人能够为你所在的机构做决定，除非他们了解你的机构。没有放之四海而皆准的规则。没有对所有机构都“正确”的答案。所以不要让任何人做仓促的选择。“思考（Think）”并非一个四字母的单词。

例如，一些机构看重的是一致性，并且已经有大量代码使用“`const Fred*`”了。对于他们来说，不管是否有优点，“`Fred const*`”都不是个好选择。还有很多其它的商业环境，一些倾向于“`Fred const*`”，其它则倾向于“`const Fred*`”。

采用适合你机构中普通维护程序员的写法。不是专家，不是傻瓜，而是维护代码的普通程序员。除非你决定解雇他们并雇佣新人，否则就要确保他们能够理解你的代码。根据实际情况做商业决定，而不是根据其它什么人的假设。

使用“`Fred const*`”需要克服一些惯性。现在大多数C++书籍都使用`const Fred*`，大多数程序员学C++时接触的就是这种语法，并且仍然这么用。这并非是说`const Fred*`一定对你的机构好。但在更改（这种风格）期间，和/或在招收新人时，的确可能会引起一些混乱。一些机构认为用`Fred const*`带来的好处更大，其它机构则不这么认为。

另一个警告：如果决定用`Fred const*`，确保采取措施使人们不会误写成语义不同但语法相似的“`Fred* const x`”。这两者虽然第一眼看上去非常相似，但含义完全不同。

18.10 什么是“ `const` 成员函数”？

是指仅查看（而不改变）对象的成员函数。

`const` 成员函数会在紧跟函数参数列表的后面跟一个 `const` 关键字。有 `const` 后缀的成员函数被称作“ `const` 成员函数”或者是“查看函数”（inspector）。没有 `const` 后缀的成员函数被称作“非 `const` 函数”或“变更函数”（mutator）。

```
class Fred {
public:
  void inspect() const;   // 该成员保证不修改*this
  void mutate();          // 该成员可能会修改*this
};

void userCode(Fred& changeable, const Fred& unchangeable)
{
  changeable.inspect();   // 正确：没有修改一个可修改对象
  changeable.mutate();    // 正确：修改一个可修改对象

  unchangeable.inspect(); // 正确：没有修改一个不可修改对象。
  unchangeable.mutate();  // 错误：试图修改一个不可修改对象。
}
```

`unchangeable.mutate()` 这个错误在编译期被发现。 `const` 不会有运行时的时空效率损失。

在 `inspect()` 成员函数后面的 `const` 后缀表示不会改变对象的（调用方可见的）抽象状态。这并非保证不改变对象的“底层二进制位”。C++编译器不允许将其解释为“按位”（不变），除非能解决别名问题，而别名问题一般无法解决（即可能存在会修改对象状态的非 `const` 别名）。另外一个对这种别名问题的（重要）认识是：用一根“指向 `const` 对象的指针”并不能保证对象不改变，它只是保证对象不会通过该指针被改变。

18.11 返回引用的成员函数和 `const` 成员函数之间有什么联系？

如果想要从一个查看函数中返回 `this` 对象的引用，那么应该返回指向 `cosnst` 对象的引用，即 `const X&` 。

```
class Person {
public:
  const std::string& name_good() const;  ← 正确：调用者不能修改name
  std::string& name_evil() const;        ← 错误：调用者能够修改name...
};

void myCode(const Person& p)  ← 这里保证不会修改Person 对象...
{
  p.name_evil() = "Igor";     ← ...但还是修改了！！
}
```

好消息是当你犯这种错误时，编译器通常能够发现。尤其是如果不小心返回了`this`对象的非`const`引用，例如上面的`Person::name_evil()`，编译器在编译这个成员函数时，通常能够发现并给出一条编译错误。

坏消息是编译器并不能发现所有这种错误：在一些情况下编译器无法产生一条错误消息。

最后：你需要思考，并记住本FAQ所述的原则。如果你通过引用返回的对象在逻辑上是`this`对象的一部分，而不管其是否在物理上放在了`this`对象内，那么`const`方法应该返回`const`引用或直接按值返回。（`this`对象的“逻辑”部分与对象的“抽象状态”相关。请参阅前一个FAQ。）

18.12 “`const`重载”是做什么用的？

当一个查看函数和一个变更函数名字相同，且参数个数与类型也相同时就有用了——即两者的不同之处仅在于一个有`const`另一个没有`const`。

`const`重载的一个常见应用是下标运算符。通常应该尽量使用标准模板容器，例如`std::vector`，但有时会需要在自己的类中支持下标运算符。一个经验法则是：下标运算符通常成对出现。

```
class Fred { ... };

class MyFredList {
public:
  const Fred& operator[] (unsigned index) const;  ←下标运算符通常成对出现
  Fred&       operator[] (unsigned index);        ←下标运算符通常成对出现...
};
```

当对一个非`const`的`MyFredList`对象使用下标运算符时，编译器会调用非`const`的下表运算符。因为返回的是一个普通`Fred&`，所以能够查看或修改对应的`Fred`对象。例如，假设`Fred`类有一个查看函数`Fred::inspect() const`和一个变更函数`Fred::mutate()`：

```
void f(MyFredList& a)  ← MyFredList是非const的
{
  // 可以调用不修改a[3]处Fred对象的方法：
  Fred x = a[3];
  a[3].inspect();

  // 可以调用修改a[3]处Fred对象的方法:
  Fred y;
  a[3] = y;
  a[3].mutate();
}
```

但是，当对一个`const`的`MyFredList`对象使用下标运算符时，编译器会调用`const`的下标运算符。因为会返回`const Fred&`，所以可以查看对应的`Fred`对象而不能修改它。

```
void f(const MyFredList& a)  ← MyFredList是const的
{
  // 可以调用不修改a[3]处Fred对象的方法:
  Fred x = a[3];
  a[3].inspect();

  // 错误（很幸运！）：试图改变a[3]出的Fred对象:
  Fred y;
  a[3] = y;         ← 幸运的是编译器在编译时发现了这个错误。
  a[3].mutate();  ← 幸运的是编译器在编译时发现了这个错误。
}
```

在以下FAQ中演示了针对下标运算符和函数调用运算符的`const`重载：[13.10], [16.17], [16.18], [16.19]和[35.2]

当然除了下标运算符，其它函数也可以进行`const`重载。

18.13 如果我想让一个 `const` 成员函数对数据成员做“不可见”的修改，应该怎么办？

用`mutable`（或者实在没办法了，用最后一招`const_cast`）

少数查看函数需要对数据成员做适当的修改（例如，一个`Set`对象可能想要缓存上一次查看的内容，以便下一次查看时能够提高性能）。这里“适当”的意思是，所做的修改不会从对象的接口上反映到外部（否则该成员函数就应该是一个变更函数，而不是查看函数了）。

这时，需要修改的数据成员应标记为`mutable`（把`mutable`关键字放在数据成员的声明前；即和`const`的位置一样）。这就通知编译器说这个数据成员允许在`const`成员函数中被修改。如果编译器不支持`mutable`关键字，那么可以通过`const_cast`去除掉`this`的`const`（但是记着读下面的注意事项）。例如在`Set::lookup() const`中，可以这么写：

```
Set* self = const_cast<Set*>(this);
  // 在这么做之前，记着读下面的**注意事项**
```

然后，`self`和`this`内容一样（即`self == this`为真），但`self`类型是`Set*`而不是`const Set*`（技术上来讲，是`const Set* const`，不过最右边的`const`与这里的问题无关）。因此可以使用`self`来修改`this`所指向的对象。

注意：`const_cast`可能会导致一种非常罕见的错误。这个错误仅在三件很少见的事情同时发生时出现：数据成员本应该是`mutable`（例如上面所说的），编译器不支持`mutable`，并且对象原本就定义为`const`（不是通过一根指向`const`对象的指针来访问的普通`const`对象）。虽然这种组合非常罕见，甚至永远不会发生，但如果真的发生了，那么这种代码可能就不能正常运行（标准说这种行为是未定义的）。

如果想要用 `const_cast` ，那么应该用 `mutable` 替代。换句话说，如果需要修改一个对象的成员，而又是通过指向 `const` 对象的指针来访问这个对象，那么最安全和最简单的做法就是给该成员的声明前加上 `mutable` 。如果你确信实际对象不是 `const` 的（例如能够确定对象是像这样声明的： `Set s;` ），那么也可以用 `const_cast` 。但如果对象本身就是 `const` 的（例如可能声明为： `const Set s;` ），那么就应该用 `mutable` 而不是 `const_cast` 。

请不要告诉我说Y编译器的X版本在Z机器上允许修改 `const` 对象的非 `mutable` 成员。我不管这个——根据标准这是错误的，如果换一个编译器，甚至是同一编译器的不同版本（升级版），你的代码就可能会失败。不要这么做。用 `mutable` 吧。

18.14 `const_cast` 会导致无法优化么？

在理论上是的；在实际中不会。

即使语言本身禁止了 `const_cast` ，要想在调用 `const` 成员函数时避免读写寄存器的唯一办法是解决别名问题（即要证明没有其它指向该对象的非 `const` 指针）。这只有在极少数情况下才能办到（当在调用 `const` 成员函数时构造对象，所有在构造对象和调用 `const` 成员函数之间调用的非 `const` 成员函数是静态绑定的，并且所有这些调用包括构造函数都是内联的，同时构造函数调用的任何成员函数也要是内联的）。

18.15 当我用 `const int*` 指向一个 `int` 后，为什么编译器还允许我修改这个 `int` ？

因为“ `const int* p` ”意思是“ `p` 保证不会修改 `*p` ”，而不是说“ `*p` 保证不变”。

用 `const int*` 指向一个 `int` ，不会使这个 `int` 变为 `const` 。 `int` 不会通过 `const int*` 被修改，但如果有另外一个 `int*` （注意没有 `const` ）指向该 `int` （这就是“别名”），那么这个 `int*` 指针可以用来修改 `int` 。例如：

```
void f(const int* p1, int* p2)
{
  int i = *p1;          // 获得*p1的（原始）值*p1
  *p2 = 7;              // 如果p1 == p2，这就会修改*p1。
  int j = *p1;          // 获取*p1（可能是更新后）的值。
  if (i != j) {
    std::cout << "*p1 changed, but it didn't change via pointer p1!\n";
    assert(p1 == p2);  // 这是*p1可能会变化的唯一办法。
  }
}

int main()
{
  int x = 5;
  f(&x, &x);            // 这完全合法（甚至是合情理的！）...
}
```

注意 `main()` 和 `f(const int*, int*)` 可能是在不同的编译单元中，并且不是在同一天编译的。因此，编译器就无法在编译时发现别名。因此无法在语言中禁止这种事情。实际上，我们甚至不想添加这样一个规则。因为一般来说，允许很多指针指向同一个对象，这是一个功能。当指针保证说不去修改所指内容时，这只是该指针作出的保证，而不是内容所做的保证。

18.16 “ `const Fred* p` ”的意思是 `*p` 不会改变么？

不是！（这个与int指针的别名问题相关）

“ `const Fred* p` ”意思是不能通过指针 `p` 来修改 `Fred` ，但有可能不经过 `const` （例如一个非 `const` 指针 `Fred*` ），而是通过其它途径来访问 `object` 。例如，如果有两根指针“ `const Fred* p` ”和“ `Fred* q` ”都指向同一个 `Fred` 对象（别名），那么指针 `q` 可以用来修改 `Fred` 对象，但指针 `p` 不能。

```
class Fred {
public:
  void inspect() const;   // const成员函数
  void mutate();          // 非const成员函数
};

int main()
{
  Fred f;
  const Fred* p = &f;
        Fred* q = &f;

  p->inspect();    // 可以：不改变*p_
  p->mutate();     // 错误：不能通过p来修改*p

  q->inspect();    // 可以：允许使用q来查看对象
  q->mutate();     // 可以：允许使用q来修改对象

  f.inspect();     // 可以：允许使用f来查看对象
  f.mutate();      _// 可以：允许使用f来修改对象...
```

18.17 当把 `Foo**` 转换成 `const Foo**` 时为什么会出错？

因为把 `Foo**` 转换成 `const Foo**` 是非法且危险的。

C++允许 `Foo*` 到 `const Foo*` 的转换（这是安全的）。但如果想要将 `Foo**` 隐式转换成 `const Foo**` 则会报错。

这么做的原因如下所示。但首先，这里有个最普通的解决办法：只要把 `const Foo**` 改成 `const Foo* const*` 就可以了。

```
class Foo { /* ... */ };

void f(const Foo** p);
void g(const Foo* const* p);

int main()
{
  Foo** p = /*...*/;
  ...
  f(p);  // 错误：将Foo**转换成const Foo**是非法且邪恶的
  g(p);  // 可以：将Foo**转换成const Foo* const*是合法且合理的...
}
```

之所以 `Foo**` 到 `const Foo**` 的转换是危险的，是因为这会使你没有经过转换就在不经意间修改了 `const Foo` 对象。

```
class Foo {
public:
  void modify();  // 修改this对象
};

int main()
{
  const Foo x;
  Foo* p;
  const Foo** q = &p;  // 这时q指向p；（幸亏）这是个错误。
  *q = &x;             // 这时p指向x
  p->modify();         // 啊：修改了const Foo！！...
}
```

记住：请不要用指针转换绕过这里。别这么做就是了！

[19] 继承 — 基础

FAQs in section [19]:

19.1 对于C++，继承是否重要？

是。

继承是面向对象编程和抽象数据类型（ADT）编程的区分标志

19.2 何时该使用继承？

作为一种特化的机制。

人们抽象事物有两种角度："部分"和"种类"。Ford Taurus是一种（is-a-kind-of-a）汽车，并且 Ford Taurus 有（has-a）引擎，轮胎等。"部分"层次已经随着ADT风格而成为软件系统的一部分。继承则增加了另一种分解的角度。

19.3 在C++中如何表达继承？

通过 `:public` 语法：

```
class Car : public Vehicle {
public:
  // ...
};
```

我们有几种方式声明以上的关系：

- `Car` 是 "一种"（"a kind of a"）`Vehicle（交通工具）`
- `Car` 起源于（"derived from"）`Vehicle`

- `Car` 是一种特殊化的（"a specialized"） `Vehicle`
- `Car` 是 `Vehicle` 的一个子类（"subclass"）
- `Car` 是 `Vehicle` 的一个派生类（“derived class”）
- `Vehicle` 是 `Car` 的基类（"base class"）
- `Vehicle` 是 `Car` 的超类（"superclass"）(这在C++ 社群中不常用)

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

19.4 将一个派生类型的指针转换成它的基类型可以吗？

可以。

派生类对象是基类对象的一种。因此从派生类指针到基类指针的转换是非常安全的，并且始终会发生。例如，如果有一个 car 类型的指针，而实际上指向了 vehicle，这种从 `Car*` 到 `Vehicle*` 的转换是非常安全的和常规的：

```
void f(Vehicle* v);
void g(Car* c) { f(c); }  // 非常安全; 不用转换
```

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

19.5 `public:` , `private:` 和 `protected:` 有什么不同？

- 在类的 `private:` 节中声明的成员（无论数据成员或是成员函数）仅仅能被类的成员函数和友元访问。
- 在类的 `protected:` 节中声明的成员（无论数据成员或是成员函数）仅仅能被类的成员函数，友元以及子类的成员函数和友元访问。
- 在类的 `public:` 节中声明的成员（无论数据成员或是成员函数）能被任何人访问。

19.6 为什么派生类不能访问基类的 `private:` 成员？

为了使派生类在将来基类改变时不受影响。

派生类无法访问基类的私有成员。这样在对基类私有成员作任何改变时，就有效地锁定了派生类。

19.7 如何才能在改变类的内在部分时，保护其派生类不被破坏？

类有两套截然不同的接口，它们分别面向两个截然不同的客户：

- 有为无关类服务的 `public:` 接口
- 有为派生类服务的 `protected:` 接口

除非你期望你的所有子类全部由你自己的团队建立，否则你应该考虑让基类部分成为 `private:` ，并且用 `protected:` 来内联供子类访问基类私有数据的访问函数。使用这种方法，私有部分可以被改变，但是派生类的代码不会被破坏（除非你改变了 `protected` 的访问函数）。

[20] 继承 — 虚函数

FAQs in section [20]:

- [20.1] 什么是“虚成员函数”？
- [20.2] C++ 怎样同时实现动态绑定和静态类型？
- [20.3] 虚成员函数和非虚成员函数调用方式有什么不同？
- [20.4] 析构函数何时该时虚拟的？
- [20.5] 什么是“虚构造函数(`virtual` constructor)”？

20.1 什么是“虚成员函数”？

从面向对象观点来看，它是 C++ 最重要的特征：[6.8], [6.9].

虚函数允许派生类取代基类所提供的实现。编译器确保当对象为派生类时，取代者（译注：即派生类的实现）总是被调用，即使对象是使用基类指针访问而不是派生类的指针。这样就允许基类的算法被派生类取代，即使用户不知道派生类的细节。

派生类可以完全地取代基类成员函数（覆盖(override)），也可以部分地取代基类成员函数（增大(augment)）。如果愿意的话，后者由派生类成员函数调用基类成员函数来完成。

20.2 C++ 怎样同时实现动态绑定和静态类型？

当你有一个对象的指针，而对象实际是该指针类型的派生类（例如：一个 `Vehicle*` 指针实际指向一个Car 对象）。由此有两种类型：指针的（静态）类型（在此是 `Verhicle` ），和指向的对象的（动态）类型（在此是Car）。

静态类型意味着成员函数调用的合法性被尽可能早地检查：编译器在编译时。编译器用指针的静态类型决定成员函数调用是否合法。如果指针类型能够处理成员函数，那么指针所指对象当然能很好的处理它。例如，如果 `Vehicle` 有某个成员函数，则由于 `Car` 是一种 `Vehicle` ，那么 `Car` 当然也有该成员函数。

动态绑定意味着成员函数调用的代码地址在最终时刻才被决定：基于运行时的对象动态类型。因为绑定到实际被调用的代码这个过程是动态完成的（在运行时），所以被称为“动态绑定”。动态绑定是虚函数导致的结果之一。

20.3 虚成员函数和非虚成员函数调用方式有什么不同？

非虚成员函数是静态确定的。也就是说，该成员函数（在编译时）被静态地选择，该选择基于指象对象的指针（或引用）的类型。

相比而言，虚成员函数是动态确定的（在运行时）。也就是说，成员函数（在运行时）被动态地选择，该选择基于对象的类型，而不是指向该对象的指针/引用的类型。这被称作“动态绑定”。大多数的编译器使用以下的一些的技术：如果对象有一个或多个虚函数，编译器将一个隐藏的指针放入对象，该指针称为“virtual-pointor”或“v-pointer”。这个v-pointer指向一个全局表，该表称为“虚函数表（virtural-table）”或“v-table”。

编译器为每个含有至少一个虚函数的类创建一个v-table。例如，如果 `Cirle` 类有虚函数d `draw()` 、 `move()` 和 `resize()` ，那么将有且只有一个和Cricle类相关的v-table，即使有一大堆Circle对象。并且每个 `Circle` 对象的 v-poiner将指向 `Circle` 的这个 v-table。该 v-table自己有指向类的各个虚函数的指针。例如， `Circle` 的v-table 会有三个指针：一个指向 `Circle::draw()` ，一个指向 `Circle::move()` ，还有一个指向 `Circle::resize()` 。

在分发一个虚函数时，运行时系统跟随对象的 v-pointer找到类的 v-table，然后跟随v-table中适当的项找到方法的代码。

以上技术的空间开销是存在的：每个对象一个额外的指针（仅仅对于需要动态绑定的对象），加上每个方法一个额外的指针（仅仅对于虚方法）。时间开销也是有的：和普通函数调用比较，虚函数调用需要两个额外的步骤（得到v-pointer的值，得到方法的地址）。由于编译器在编译时就通过指针类型解决了非虚函数的调用，所以这些开销不会发生在非虚函数上。

注意：由于没有涉及诸如多继承，虚继承，RTTI等内容，也没有涉及诸如page fault，通过指向函数的指针调用函数等空间/时间论的内容，所以以上讨论是相当简单的。如果你想知道其他的内容，请询问 `comp.lang.c++` ；而不要给我发E-MAIL！

20.4 析构函数何时该时虚拟的？

当你可能通过基类指针删除派生类对象时。

虚函数绑定到对象的类的代码，而不是指针/引用的类。如果基类有虚析构函数， `delete basePtr` 时（译注：即基类指针）， `*basePtr` 的对象类型的析构函数被调用，而不是该指针的类型的析构函数。这通常是一件好事情。

TECHNO-GEEK WARNING; PUT YOUR PROPELLER HAT ON. 从技术上来说，如果你打算允许其他人通过基类指针调用对象的析构函数（通过 `delete` 这样做是正常的），并且被析构的对象是有重要的析构函数的派生类的对象，就需要让基类的析构函数成为虚拟的。如果一个类有显式的析构函数，或者有成员对象，该成员对象或基类有重要的析构函数，那么这个类就有重要的析构函数。（注意这是一个递归的定义（例如，某个具有重要析构函数的类，它有一个成员对象（它有基类（该基类有成员对象（它有基类（该基类有显式的析构函数））））） *END TECHNO-GEEK WARNING; REMOVE YOUR PROPELLER HAT*

如果你对以上的规则理解有困难，试试这个简单的：类应该有虚析构函数，除非这个类没有虚函数。原理：如果有虚函数，说明你想通过基类指针来使用派生对象，并且你所可能做的事情之中，可能包含了调用析构函数（通常通过 `delete` 隐含完成）。一旦你在类中加上了一个虚函数，你就已经需要为每一个对象支付空间代价（每个对象一个指针；注意这是理论上的编译器特性；实际上每个编译器都是这样做的），所以这时使析构函数成为虚拟的通常不会额外付出什么。

20.5 什么是“虚构造函数(`virtual` constructor)”？

一种允许你做一些 C++ 不直接支持的事情的用法。

你可能通过虚函数 `virtual` `clone()`（对于拷贝构造函数）或虚函数 `virtual` `create()`（对于默认构造函数），得到虚构造函数产生的效果。

```
class Shape {
public:
  virtual ~Shape() { }                 // 虚析构函数
  virtual void draw() = 0;             // 纯虚函数
  virtual void move() = 0;
  // ...
  virtual Shape* clone()  const = 0;   // 使用拷贝构造函数_
  virtual Shape* create() const = 0;   // 使用默认构造函数
};

class Circle : public Shape {
public:
  Circle* clone()  const { return new Circle(*this); }
  Circle* create() const { return new Circle();      }
  // ...
};
```

在 `clone()` 成员函数中，代码 `new Circle(*this)` 调用 `Circle` 的拷贝构造函数来复制 `this` 的状态到新创建的 `Circle` 对象。在 `create()` 成员函数中，代码 `new Circle()` 调用 `Circle` 的默认构造函数。

用户将它们看作“虚构造函数”来使用它们：

```
void userCode(Shape& s)
{
  Shape* s2 = s.clone();
  Shape* s3 = s.create();
  // ...
  delete s2;    // 在此处，你可能需要虚析构函数
  delete s3;
}
```

这个函数将正确工作，而不管 `Shape` 是一个 `Circle`，`Square`，或是其他种类的 `Shape`，甚至它们还并不存在。

注意：成员函数`Circle`'s `clone()`的返回值类型故意与成员函数`Shape`'s `clone()`的不同。这种特征被称为“协变的返回类型”，该特征最初并不是语言的一部分。如果你的编译器不允许在`Circle`类中这样声明`Circle* clone() const`（如，提示“The return type is different”或“The member function's type differs from the base class virtual function by return type alone”），说明你的编译器陈旧了，那么你必须改变返回类型为`Shape*`。

[21] 继承 — 适当的继承和可置换性

FAQs in section [21]:

- [21.1] 我应该隐藏基类的公有成员函数吗？
- [21.2] `Derived* —> Base*` 可以很好地工作; 为什么 `Derived** —> Base**` 不行？
- [21.3] parking-lot-of-Car（停车场）是一种 parking-lot-of-Vehicle（交通工具停泊场）吗？
- [21.4] `Derived` 数组是一种 `Base` 数组吗？
- [21.5] 派生类数组(array-of- `Derived`)"不是一种"基类数组(array-of- `Base`)是否意味着数组不好？
- [21.6] `Circle` （圆）是一种 `Ellipse` （椭圆）吗？
- [21.7] 对于"圆是/不是一种椭圆"这个两难问题，有其它说法吗？
- [21.8] 但我是数学博士，我相信圆是一种椭圆！这是否意味着Marshall Cline是傻瓜？或者C++是傻瓜？或者OO是傻瓜？
- [21.9] 也许椭圆应该从圆继承？
- [21.10] 但我的问题与圆和椭圆无关，这种无聊的例子对我有什么好处？

21.1 我应该隐藏基类的公有成员函数吗？

不要，不要，不要这样做。永远不要！

试图隐藏（消除、废除、私有化）继承而来的公有成员函数是非常常见的设计错误。通常这产生于浆糊脑袋。

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

21.2 `Derived* —> Base*` 可以很好地工作; 为什么 `Derived** —> Base**` 不行？

由于 `Derived` 对象是一种 `Base` 对象，C++允许 `Derived*` 转换成 `Base*` 。然而，将 `Derived**` 转换成 `Base**` 将产生错误。尽管这个错误不是显而易见的，这未尝不是件好事。例如，如果你能够将 `Car**` 转换成 `Vehicle**` （译注：Vehicle意为交通工具），并且如果你能同样的将 `NuclearSubmarine**` （译注：NuclearSubmarine意为核潜艇） 转换成 `Vehicle**` ，那么你可能给这两个指针赋值，并最终使 `Car*` 指针指向 `NuclearSubmarine` ：

```
class Vehicle {
public:
  virtual ~Vehicle() { }
  virtual void startEngine() = 0;
};

class Car : public Vehicle {
public:
  virtual void startEngine();
  virtual void openGasCap();
};

class NuclearSubmarine : public Vehicle {
public:
  virtual void startEngine();
  virtual void fireNuclearMissle();
};

int main()
{
  Car   car;
  Car*  carPtr = &car;
  Car** carPtrPtr = &carPtr;
  Vehicle** vehiclePtrPtr = carPtrPtr;  // 这在C++中是一个错误
  NuclearSubmarine  sub;
  NuclearSubmarine* subPtr = &sub;
  *vehiclePtrPtr = subPtr;
  // 最后这行将导致carPtr指向 sub !
  carPtr->openGasCap();  // 这将调用 fireNuclearMissle()! （译注：也就是发射核弹）
}
```

换句话说，如果从 `Derived**` 到 `Base**` 的转换是合法的，那么 `Base**` 将可能被解除引用（易变的 `Base*` ），并且 `Base*` 可能被指向不同的派生类对象，这将导致严重的国家安全问题（天知道如果你调用了 `NuclearSubmarine` （核潜艇）对象的 `openGasCap()` 成员函数会发生什么!!而你却认为这是一个 `Car` 对象!!——试一下以上的代码，看看会发生什么——大多数的编译器会调用 `NuclearSubmarine::fireNuclearMissle()` !

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

21.3 parking-lot-of-Car（停车场）是一种 parking-lot-of-Vehicle（交通工具停泊场）吗？

不。

我知道这听起来很奇怪，但这是事实。你可以将这看作为以上 FAQ的直接结论，或者你可以这样来理解：如果这个“是一种”关系成立的话，那么就可以将 parking-lot-of-Vehicle 类型的指针指向一个 parking-lot-of-Car。但是，parking-lot-of-Vehicle 有 `addNewVehicleToParkingLot(Vehicle&)` 成员函数用来向停泊场添加任何 `Vehicle` （交通工具）对象。这样将允许你在 parking-lot-of-Car（停车场）停泊一个 `NuclearSubmarine` （核潜艇）。当然，当某人认为从 parking-lot-of-Car 删除一个 `Car` 对象，而实际是一个 `NuclearSubmarine` 时，他会非常惊讶。

用另一种方法阐述这个事实：一种事物的容器不是一种任何事物的容器。也许很难接受，但这是事实。

你可以不喜欢它，但必须接受它。

我们在OO/C++训练课程使用的最后一个例子：“一袋苹果不是一袋水果”。如果一袋苹果能够被传递给一袋水果的话，就可以把香蕉放入袋中，即使它被认为里面只能放苹果！

(注意: 本 FAQ 的论述仅与公有继承（`public` inheritance）有关; 私有和保护继承并不相同)

21.4 `Derived` 数组是一种 `Base` 数组吗？

不。

这是以上FAQ的结论。不幸的是它会把你带入困境，考虑一下这个：

```
class Base {
public:
  virtual void f();             // 1
};

class Derived : public Base {
public:
  // ...
private:
  int i_;                       // 2
};

void userCode(Base* arrayOfBase)
{
  arrayOfBase[1].f();           // 3
}

int main()
{
  Derived arrayOfDerived[10];   // 4
  userCode(arrayOfDerived);     // 5
}
```

编译器会认为这是完美的类型安全。编号 5的这一行将 `Derived*` 转换为 `Base*` 。但实际上这样做是可怕的：由于 `Derived` 比 `Base` 大，在编号3的这一行的指针运算是错误的：当编译器计算 `arrayOfBase1]` 的地址时使用 `sizeof(Base)` ，而数组其实是一个 `Derived` 数组，这意味着在编号3的这一行的所计算的地址（以及之后的成员函数 `f()` 的调用）并不在任何对象的起始位置！而在 `Derived` 对象的中间。假设你的编译器使用通常的方法寻找[虚函数，那么将导致第一个 `Derived` 对象的 `int i_` 被重新解释，将它看作指向虚函数表的指针，跟随着这个“指针”（意味着我们正在访问一个随机的内存位置），并将内存中那个位置的前几个字节解释为 C++成员函数的地址，然后将它们（随机的内存地址）装载到指令寄存器并开始从那个内存区产生机器指令。发生这样情况的几率相当高。

根本问题是 C++无法区别指向事物的指针和指向事物数组的指针。自然的，C++是从C继承了这一特征。

注意：如果我们使用类似数组（array-like）的类（例如，标准库中的 `std::vector<Derived>` ）来代替原始的数组，这个问题将会被作为编译时错误找出而不是运行时的灾难。

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

21.5 派生类数组（array-of- `Derived`）“不是一种”基类数组（`array-of- Base`）是否意味着数组不好？

是的，数组很差劲。（开个玩笑）。

真诚的来说，数组和指针非常接近，并且指针很难处理。但是如果我们完全掌握了为什么从设计角度来看，以上FAQ所说的会是一个问题（例如，如果你真的知道为什么事物的容器不是一种任何事物的容器），并且你认为将维护你的代码的其他人都完全掌握这些OO的设计事实的话，那么你可以自由使用数组。但是如果你象大多数人一样的话，你应该使用诸如标准库的 `std::vector<T>` 这样的模板容器类而不是原始的数组。

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

21.6 `Circle` （圆）是一种 `Ellipse` （椭圆）吗？

如果椭圆允许改变圆率，则不是。

例如，假设椭圆有一个 `setSize(x,y)` 成员函数，并且这个成员函数允许椭圆的 `width()` 是 `x` ， `height()` 是 `y` 。在这种情况下，圆无法是一种椭圆。很简单，如果椭圆能做某些圆不能做的事，则圆不是一种椭圆。

据此推出圆和椭圆的两种（合法的）关系：

- 使圆类和椭圆类完全无关
- 使圆和椭圆都从一个基类派生，该基类是“不能执行不对称 `setSize()` 运算的椭圆”

在第一种情况下，椭圆可以从 `AsymmetricShape` （不对称图形）类派生， `setSize(x,y)` 可以在 `AsymmetricShape` 类中声明。而圆可以从有 `setSize(size)` 成员函数的 `SymmetricShape` （对称图形）类派生。

在第二种情况下， `Oval` （卵形）类可以只有 `setSize(size)` 来同时设置 `width()` 和 `height()` 的大小。椭圆和圆都继承自 `Oval` 。椭圆（但不是圆）可以增加 `setSize(x,y)` 运算（但如果 `setSize()` 成员函数名称重复，当心隐藏规则）

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

(注意: `setSize(x,y)` 并不是神圣的。依赖于你的目标，防止用户改变椭圆的尺寸也是可以的。在某些情况下，椭圆没有 `setSize(x,y)` 方法是有效的设计选择。然而这个系列的讨论是当你想为一个已存在的类建立一个派生类并且基类含有一个“无法接受”的方法时，该如何做。当然理想情形是在基类不存在时就发现这个问题。但生活并不总是理想的……)

21.7 对于“圆是/不是一种椭圆”这个两难问题，有其它说法吗？

如果你主张所有椭圆是可以被压成不对称的，并且你主张圆是一种椭圆，并且你主张圆不能被压成不对称的。无疑你必须调整（实际上是撤回）你的主张之一。由此，你要么去掉 `Ellipse::setSize(x,y)`，去掉圆和椭圆的继承关系，要么承认你的 `Circle` s（圆）不必是正圆。

这里有两个OO/C++编程新手通常会陷入的陷阱。他们会试图用代码的技巧来弥补设计的缺陷（他们会重定义 `Circle::setSize(x,y)` 来抛出异常，调用 `abort()`，取两个参数的平均数，或者什么都不做）。不幸的是，由于用户期望 `width() == x` 并且 `height() == y`，所以这些技巧会使用户惊讶。而让用户惊讶是不允许的。

如果保持“圆是一种椭圆”的继承关系对你来说非常重要，那么你只能削弱椭圆的 `setSize(x,y)` 所做的承诺。例如，你可以改变承诺为，“该城圆函数可以把 `width()` 设置为 `x` 并且/或把 `height()` 设置为 `y`，或不做什么事情”。不幸的是由于用户没有任何意义的行为可以倚靠，这样会冲淡契约。因此整个层次都变得没有价值（如果某人问你到对象能做什么，而你只能耸耸肩膀的话，你很难说服他取使用这个对象）

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

(注意: `setSize(x,y)` 并不是神圣的。依赖于你的目标，防止用户改变椭圆的尺寸也是可以的。在某些情况下，椭圆没有 `setSize(x,y)` 方法是有效的设计选择。然而这个系列的讨论是当你想为一个已存在的类建立一个派生类并且基类含有一个“无法接受”的方法时，该如何做。当然理想情形是在基类不存在时就发现这个问题。但生活并不总是理想的……)

21.8 但我是数学博士，我相信圆是一种椭圆！这是否意味着Marshall Cline是傻瓜？或者C++是傻瓜？或者OO是傻瓜？

事实上，这并不意味着这些。而是意味着你的直觉是错误的。

看，我收到并回复了大量的关于这个主题的热情的e-mail。我已经给各地上千个软件专家讲授了数百次。我知道它违背了你的直觉。但相信我，你的直觉是错误的。

眞正的问题是你的直觉中的“是一种（kind of）”的概念不符合OO中的适当的继承（学术上称为“子类型(subtyping)”）概念。派生类对象最起码必须是可以取代基类对象的。在圆/椭圆的情况下，`setSize(x,y)` 成员函数违背了这个可置换性。

你有三个选择：[1]从 `Ellipse` （椭圆）类中删除 `setSize(x,y)` 成员函数（从而废弃调用 `setSize(x,y)` 成员函数的已存在代码），[2]允许 `Circle` （圆）的高和宽不同（一个不对称的圆），或者[3]去掉继承关系。抱歉，但没有其他选择。有人提过另一个选项，让圆和椭圆都从第三个通用基类派生，但这只不过是以上选项[3]的变种罢了。

换一种说法就是，你要么使基类弱一些（在这里就是说你不能为椭圆的高和宽设置不同的值），要么使派生类强一些（在这里就是使圆同时具有对称的和不对称的能力）。当这些都无法令人满意（就如圆/椭圆例子），通常就简单的消除继承关系。如果继承关系必须存在，你只能从基类中删除变形成员函数（ `setHeight(y)` ， `setWidth(x)`， 和 `setSize(x,y)` ）

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

(注意: `setSize(x,y)` 并不是神圣的。依赖于你的目标，防止用户改变椭圆的尺寸也是可以的。在某些情况下，椭圆没有 `setSize(x,y)` 方法是有效的设计选择。然而这个系列的讨论是当你想为一个已存在的类建立一个派生类并且基类含有一个“无法接受”的方法时，该如何做。当然理想情形是在基类不存在时就发现这个问题。但生活并不总是理想的……)

21.9 也许椭圆应该从圆继承？

如果圆是基类，椭圆是派生类的话，那么你会面临许多新的问题。例如，假设圆有 `radius()` 方法（译注：设置半径的成员函数）。那么椭圆也会有 `radius()` 方法，但那没有意义：一个椭圆（可能不对称）的半径是什么意思？

如果你克服这个障碍（也就是使得 `Ellipse::radius()` 返回主轴和辅轴的平均值或其它办法），那么 `radius()` 和 `area()` （译注：得到面积的成员函数）之间的关联就会有问题。比如，假设圆有 `area()` 方法返回的是3.14159乘以 `radius()` 返回值的平方。而 `Ellipse::area()` 将不会返回椭圆的眞实面积，否则你必须记住让 `radius()` 返回符合上述公式的某个值。

即使你克服了这个问题（也就是使得 `Ellipse::radius()` 返回了椭圆的面积除以pi的平方根），你还要应付 `circumference()` 方法（译注：计算周长的成员函数）。比如，假设圆有 `circumference()` 方法返回2乘以pi乘以 `radius()` 的返回值。现在你的麻烦是：对于椭圆没有办法两碗水端平了：椭圆类不得不在面积，或者周长，或者两者的计算上撒谎。（译注：对于椭圆，面积和周长的计算无法同时得到正确答案，因为它们都使用了 `radius()` 的返回值，而它们对于 `radius()` 的返回值的要求却不相同， `radius()` 无法同时满足它们的需要）

底线：只要派生类遵守基类的承诺，你就可以使用继承。而不能仅仅因为你感觉上象继承或仅仅因为你想使得代码被重用就使用继承。只有在(a)派生类的方法能遵守基类所做的所有承诺，并且(b)用户不会被你搞糊涂，并且(c)使用继承能明显获得实在的时间上的，金钱上的或

风险上的改进时，才应该使用继承。

21.10 但我的问题与圆和椭圆无关，这种无聊的例子对我有什么好处？

啊，有点小误会。你认为圆/椭圆例子是无聊的，但实际上，你的问题和它是同性质的。

我不在意你的继承问题是什么，但所有（是的，所有）不良的继承都可以归结为“圆不是一种椭圆”的例子。

这就是为什么：不良的继承总有一个有额外能力（经常是一个或两个额外的成员函数；有时是一个或多个成员函数给出的承诺）的基类，而派生类却无法满足它。你要么使基类弱一些，派生类强一些，要么消除继承关系。我见过很多很多很多不良的继承方案，相信我，它们都可以归结为圆/椭圆的例子。

因此，如果你真的理解了圆/椭圆的例子，你就能找出所有的不良继承。如果你没有理解圆/椭圆问题，那么你很可能犯一些严重的并且昂贵的继承错误。

令人忧伤，但是真的。

(注意: 本 FAQ 的论述仅与公有继承（ `public` inheritance）有关; 私有和保护继承并不相同)

[22] 继承 — 抽象基类(ABCs)

FAQs in section [22]:

22.1 将接口和实现分离的作用是什么？

接口是公司最有价值的资源。设计接口比用一堆类来实现这个接口更费时间。而且接口需要更昂贵的人力的时间。

既然接口如此有价值，它们应该被保护，以免因为数据结构和其他实现的改变而被破坏。因此，应该将接口和实现分离。

22.2 在C++中如何分离接口和实现（就象 Modula-2）？

使用ABC。（译注：即抽象基类 abstract base class）

22.3 什么是 ABC?

抽象基类（abstract base class）。

在设计层次，抽象基类（ABC）对应于抽象概念。如果你问一个机修工他是否修理交通工具，他可能想知道你所说的是哪种交通工具。他不修理航天飞机、远洋轮、自行车或核潜艇。问题在于“交通工具”是一个抽象概念（例如，除非你知道你要建造哪种交通工具，否则你无法建造一个“交通工具”）。在C++中，`Vehicle`（交通工具）类是一个ABC，而`Bicycle`（自行车），`SpaceShuttle`（航天飞机）等则是派生类（`OceanLiner(远洋轮)`是一种`Vehicle`）。在真实世界的OO中，ABC无处不在。

在程序语言层次上，抽象基类（ABC）是有一个或多个纯虚成员函数的类。无法建立抽象基类的对象（实例）。

22.4 什么是“纯虚”成员函数？

将普通类变成抽象基类（也就是ABC）的成员函数。通常只在派生类中实现它。

某些成员函数只在概念中存在，而没有合理的定义。例如，假设我让你在坐标 `(x,y)` 处画一个图形，大小为 7。你会问我：“我应该画哪种图形？”（圆，矩形，六边形等，画法都不同）。在C++中，我们必须指出 `draw()` 成员函数的实在物（由此用户才能在有一个 `Shape*` 或者 `Shape&` 的时候调用它），但我们认识到，在逻辑上，它只能在子类中被定义：

```
class Shape {
public:
  virtual void draw() const = 0;  // = 0 表示它是 "纯虚" 的
  // ...
};
```

这个纯虚函数使 `Shape` 成为了ABC（抽象基类）。如果你愿意，你可以将“ `= 0;` ”语法看作为代码位于NULL指针处。因此 `Shape` 向它的用户承诺了一个服务，然而 `Shape` 无法提供任何代码来实现这个承诺。这样做使得即使基类没有足够的信息来实际定义成员函数时，也强制了任何由 `Shape` 派生的具体类的对象须给出成员函数。

注意，为纯虚函数提供一个实现是可能的，但是这样通常会使初学者糊涂，并且最好避免这样，直到熟练之后。

22.5 如何为包含指向（抽象）基类的指针的类定义拷贝构造函数或赋值操作符？

如果类拥有被（抽象）基类指针指向的对象，则在（抽象）基类中使用虚构造函数用法](virtual-functions.html#[20.5])。就如同一般用法一样，在基类中声明一个[纯虚方法 `clone()` ：

```
class Shape {
public:
  // ...
  virtual Shape* clone() const = 0;   // 虚拟（拷贝）构造函数
  // ...
};
```

然后在每个派生类中实现 `clone()` 方法：

```
class Circle : public Shape {
public:
  // ...
  virtual Shape* clone() const { return new Circle(*this); }
  // ...
};

class Square : public Shape {
public:
  // ...
  virtual Shape* clone() const { return new Square(*this); }
  // ...
};
```

现在假设每个 `Fred` 对象有一个 `Shape` 对象。`Fred` 对象自然不知道 `Shape` 是圆还是矩形还是......。`Fred` 的拷贝构造函数和赋值操作符将调用 `Shape` 的 `clone()` 方法来拷贝对象：

```
class Fred {
public:
  Fred(Shape* p) : p_(p) { assert(p != NULL); }   // p must not be NULL
 ~Fred() { delete p_; }
  Fred(const Fred& f) : p_(f.p_->clone()) { }
  Fred& operator= (const Fred& f)
    {
      if (this != &f) {              // 检查自赋值_
        Shape* p2 = f.p_->clone();   // Create the new one FIRST...
        delete p_;                   // ...THEN delete the old one
        p_ = p2;
      }
      return *this;
    }
  // ...
private:
  Shape* p_;
};
```

[23] 继承 — 你所不知道的

FAQs in section [23]:

23.1 基类的非虚函数调用虚函数可以吗？

可以。有时（并非总是!）这是一个好主意。例如，假设所有`Shape`（图形）对象有一个公共的打印算法。但这个算法依赖于它们的面积并且它们都有不同的方法来计算面积。在这种情况下，`Shape`的`area()`方法（译注：得到`Shape`面积的成员函数）必须是virtual的（可能是纯虚(pure-virtual)的），但`Shape::print()`可以在`Shape`中被定义为非虚(non-virtual)的，前提是所有派生类不会需要不同的打印算法。

```
#include "Shape.hpp"

void Shape::print() const
{
    float a = this->area();  // area() 为纯虚
    // ...
}
```

23.2 上面那个FAQ让我糊涂了。那是使用虚函数的另一种策略吗？

是的，那是不同的策略。是的，那的确是使用虚函数的两种不同的基本方法：

1. 假设你遇到了上一个FAQ所描述的情况：每一个派生类都有一个结构完全一样，只有一小块不同的方法。因此算法是相同的，但实质不相同。在这种情况下，你最好在基类写一个全面的算法作为`public:`方法（有时是非虚的），然后在派生类中写那不同的一小块。这一小块在基类中声明（通常是`protected:`的，纯虚的，当然至少是`virtual`的），并且最终在每个派生类中被定义。这种情况下最紧要的问题是包含全面的算法的`public:`方法是否应该是`virtual`的。答案是，如果你认为某些派生类可能需要覆盖它，就让它成为`virtual`的。
2. 假设你遇到了上一个FAQ完全相反的情况，每一个派生类都有一个结构完全不同，但有

> 一小块的大多数（如果不是全部的话）相同的方法。在这种情况下，你最好将全面的算法放在最终在派生类中定义的`public:` `virtual`之中，并且将一小块可以被只写一次的公共代码（避免代码重复）隐藏在某处（任何地方！）。一般放在基类的`protected:`部分，但不是必须的，也可能不是最好的。找个地方隐藏它们就行了。注意，由于`public:`用户不需要/不想做它们做的事情，如果在基类中隐藏它们，通常应该使它们是`protected:`的。假定它们是`protected:`的，那么可能不应该是`virtual`的：如果派生类不喜欢它们之一的行为，可以不必调用这个方法。

强调一下，以上列表中的是“既/又”情况，而不是“二者选一”的。换句话说，在任何给定的类上，不必在两种策略中选择。既有一个符合策略 #1 的方法`f()`，又有一个符合策略 #2 的方法`g()`是非常正常的。换句话说，在同一个类中，有两种策略同时工作是非常正常的。

23.3 当基类构造函数调用虚函数时，为什么不调用派生类重写的该虚函数？

当基类被构造时，对象还不是一个派生类的对象，所以如果 `Base::Base()` 调用了虚函数 `virt()`，则 `Base::virt()` 将被调用，即使 `Derived::virt()`（译注：即派生类重写的虚函数）存在。

同样，当基类被析构时，对象已经不再是一个派生类对象了，所以如果 `Base::~Base()` 调用了`virt()`，则 `Base::virt()`得到控制权，而不是重写的 `Derived::virt()` 。

当你可以想象到如果 `Derived::virt()` 涉及到派生类的某个成员对象将造成的灾难的时候，你很快就能看到这种方法的明智。详细来说，如果 `Base::Base()`调用了虚函数 `virt()`，这个规则使得 `Base::virt()`被调用。如果不按照这个规则，`Derived::virt()`将在派生对象的派生部分被构造之前被调用，此时属于派生对象的派生部分的某个成员对象还没有被构造，而 `Derived::virt()`却能够访问它。这将是灾难。

23.4 派生类可以重置（“覆盖”）基类的非虚函数吗？

合法但不合理。

有经验的 C++ 程序员有时会重新定义非虚函数（例如，派生类的实现可能可以更有效地利用派生类的资源），或者为了回避隐藏规则。即使非虚函数的指派基于指针/引用的静态类型而不是指针/引用所指对象的动态类型，但其客户可见性必须是一致的。

23.5 “`Warning: Derived::f(float) hides Base::f(int)`”是什么意思？

意思是：你要完蛋了。

你所处的困境是：如果基类声明了一个成员函数 `f(int)` ，并且派生类声明了一个成员函数 `f(float)` （名称相同，但参数类型和/或数量不同），那么 `Base` 的 `f(int)` 被隐藏（hidden）而不是被重载（overloaded）或被重写（overridden）（即使 基类的 `f(int)` 是虚拟的）

以下是你如何摆脱困境：派生类必须有一个被隐藏成员函数的 `using` 声明，例如：

```
class Base {
public:
  void f(int);
};

class Derived : public Base {
public:
  using Base::f;    // This un-hides Base::f(int)
  void f(double);
};
```

如果你的编译器不支持 `using` 语法，那么就重新定义基类的被隐藏的成员函数，即使它们是非虚的。一般来说这种重定义只不过使用 `::` 语法调用了基类被隐藏的成员函数，如，

```
class Derived : public Base {
public:
  void f(double);
  void f(int i) { Base::f(i); }  // The redefinition merely calls Base::f(int)
};
```

23.6 "virtual table" is an unresolved external 是什么意思？

如果你得到一个连接错误" `Error: Unresolved or undefined symbols detected: virtual table for class Fred` "，那么可能是你在 `Fred` 类中有一个未定义的虚成员函数。

编译器通常会为含有虚函数的类创建一个称为“虚函数表”的不可思议的数据结构（这就是它如何处理动态绑定的）。通常你根本不必知道它。但如果你忘了为 `Fred` 类定义一个虚函数，则有时会得到这个连接错误。

许多编译器将这个不可思议的“虚函数表”放进定义类的第一个非内联虚函数的编辑单元中。因此如果 `Fred` 类的第一个非内联虚函数是 `wilma()` ，那么编译器会将 `Fred` 的虚函数表放在 `Fred::wilma()` 所在的编辑单元里。不幸的是如果你意外的忘了定义 `Fred::wilma()` ，那么你会得到一个" `Fred` 's virtual table is undefined"（ `Fred` 的虚函数表未定义）的错误而不是“ `Fred::wilma()` is undefined”（ `Fred::wilma()` 未定义）。

[24] 继承 — 私有继承和保护继承

FAQs in section [24]:

24.1 如何表示“私有继承”？

用 `: private` 代替 `: public`，例如

```
class Foo : private Bar {
public:
  // ...
};
```

24.2 私有继承和组合(composition)有什么类似？

私有继承是组合的一种语法上的变形（聚合或者 “有一个”）

例如，“汽车有一个(has-a)引擎”关系可以用单一组合表示为：

```
class Engine {
public:
  Engine(int numCylinders);
  void start();                 // Starts this Engine
};

class Car {
public:
  Car() : e_(8) { }             // Initializes this Car with 8 cylinders
  void start() { e_.start(); }  // Start this Car by starting its Engine
private:
  Engine e_;                    // Car has-a Engine
};
```

同样的“有一个”关系也能用私有继承表示：

```
class Car : private Engine {    // Car has-a Engine
public:
  Car() : Engine(8) { }         // Initializes this Car with 8 cylinders
  using Engine::start;          // Start this Car by starting its Engine
};
```

两种形式有很多类似的地方：

- 两种情况中，都只有一个 Engine 被确切地包含于 Car 中
- 两种情况中，在外部都不能将 `Car*` 转换为 `Engine*`
- 两种情况中，`Car` 类都有一个 `start()` 方法，并且都在包含的 `Engine` 对象中调用 `start()` 方法。

也有一些区别：

- 如果你想让每个 `Car` 都包含若干 `Engine`，那么只能用单一组合的形式
- 私有继承形式可能引入不必要的多重继承
- 私有继承形式允许 `Car` 的成员将 `Car*` 转换成 `Engine*`
- 私有继承形式允许访问基类的保护（`protected`）成员
- 私有继承形式允许 `Car` 重写 `Engine` 的虚函数
- 私有继承形式赋予 `Car` 一个更简洁（20个字符比28个字符）的仅通过 `Engine` 调用的 `start()` 方法

注意，私有继承通常用来获得对基类的 protected: 成员的访问，但这只是短期的解决方案（权宜之计）

24.3 我应该选谁：组合还是私有继承？

尽可能用组合，万不得已才用私有继承

通常你不会想访问其他类的内部，而私有继承给你这样的一些的特权（和责任）。但是私有继承并不有害。只是由于它增加了别人更改某些东西时，破坏你的代码的可能性，从而使维护的花费更昂贵。

当你要创建一个 `Fred` 类，它使用了 `Wilma` 类的代码，并且 `Wilma` 类的这些代码需要调用你新建的 `Fred` 类的成员函数。在这种情况下，`Fred` 调用 `Wilma` 的非虚函数，而 `Wilma` 调用（通常是纯虚函数）被 `Fred` 重写的这些函数。这种情况，用组合是很难完成的。

```
class Wilma {
protected:
  void fredCallsWilma()
    {
      std::cout << "Wilma::fredCallsWilma()\n";
      wilmaCallsFred();
    }
  virtual void wilmaCallsFred() = 0;   // 纯虚函数
};

class Fred : private Wilma {
public:
  void barney()
    {
      std::cout << "Fred::barney()\n";
      Wilma::fredCallsWilma();
    }
protected:
  virtual void wilmaCallsFred()
    {
      std::cout << "Fred::wilmaCallsFred()\n";
    }
};
```

24.4 从私有继承类到父类需要指针类型转换吗？

一般来说，不。

对于该私有继承类的成员函数或者友元来说，和基类的关系是已知的，并且这种从 `PrivatelyDer*` 到 `Base*` （或 `PrivatelyDer&` 到 `Base&` ）的向上转换是安全的，不需要也不推荐进行类型转换。

然而，对于该私有继承类（ `PrivatelyDer` ）的用户来说，应该避免这种不安全的转换。因为它基于 `PrivatelyDer` 的私有实现，它可以自行改变。

24.5 保护继承和私有继承的关系是什么？

相同点：都允许重写私有/保护基类的虚函数，都不表明派生类“是一种（a kind-of）”基类。

不同点：保护继承允许派生类的派生类知道继承关系。如此，子孙类可以有效的得知祖先类的实现细节。这样既有好处（它允许保护继承的子类使用它和保护基类的关联）也有代价（保护派生类不能在无潜在破坏更深派生类的情况下改变这种关联）。

保护继承使用 `: protected` 语法：

```
class Car : protected Engine {
public:
  // ...
};
```

24.6 私有继承和保护继承的访问规则是什么？

以这些类为例：

```
class B                    { /*...*/ };
class D_priv : private   B { /*...*/ };
class D_prot : protected B { /*...*/ };
class D_publ : public    B { /*...*/ };
class UserClass            { B b; /*...*/ };
```

子类都不能访问 `B` 的私有部分。在 `D_priv` 中，B 的公有和保护部分都是私有的。在 `D_prot` 中，`B` 的公有和保护部分都是保护的。在 `D_publ` 中，`B` 的公有部分是公有的，`B` 的保护部分是保护的（`D_publ` 是一种 `B` ）。`UserClass` 类仅仅可以访问 `B` 的公有部分。

要使 `B` 的公有成员在 `D_priv` 或 `D_prot` 中也是公有的，则使用 `B::` 前缀声明成员的名称。例如，要使 `B::f(int,float)` 成员在 `D_prot` 中公有，应该这样写：

```
class D_prot : protected B {
public:
  using B::f;  // 注意: 不是 using B::f(int,float)
};
```

[27] 编码规范

FAQs in section [18]:

27.1 有哪些好的C++编码规范？

很高兴你在这里找答案，而不仅仅是试图建立自己的编码规范。

但要注意在comp.lang.c++上有一些人对这个话题非常敏感。几乎所有的软件工程师在某个时候，都曾经被一些人利用过，这些人把编码规范当作是一种“权力游戏”。另外，有些不知道自己在说些什么的人也设定了一些C++编码规范，因此当这些标准的制定者实际写代码时，标准往往就成了只是用来观赏的东西。这些情况促使人们不相信编码规范。

很明显，问这个问题的人们是想要得到训练，因此他们并不逃避他们在知识上的欠缺。但虽然如此，在comp.lang.c++上发帖问这个问题常常会导致争吵，而不是解决方案。

Sutter和Alexandrescu对这个问题有本非常好的书叫"C++ Coding Standards"（220页，Addison-Wesley出版，2005，ISBN 0-321-11358-6）。里面提供了101条规则、指导和最佳时间。作者和编辑们提供了一些确切的材料，并且对结对审查团队很有帮助。所有这些都使得此书更有价值。值得购买。

27.2 编码规范是必需的吗？有它就够了么？

编码规范不会把一个不懂OO的程序员变得懂OO了，这需要培训和经验。编码规范的好处是，当大型机构协调不同群体的程序员时，有助于降低分化的产生。

但仅有编码规范是不够的。编码规范减少了新人的自由度，使他们不用操心一些事情，这是好的。但仅有编码规范是不够的，还需要更多实用的指南。机构需要一种一致的设计和实现的哲学。例如，是使用强类型还是弱类型？在接口中使用引用还是指针？用stream I/O还是stdio？C++代码能够调用C代码么？反过来可以么？抽象基类应该怎么使用？继承是应该采用实现方法还是特化方法？应采用何种测试策略和审查策略？接口应该统一为每个数据成员提供get()和/或set()么？接口应该是从外向内设计还是从内向外设计？错误是应该用try/catch/throw处理还是用错误码？等等。

需要有一份有关详细设计的“伪标准”。我推荐一种三头并进的方法来达到这种标准化程度：培训、指导和库。培训能够提供“强化的指示”，指导能够让OO在实际中得到应用而不仅仅是教过就没事了。而高质量的C++类库则是一种“长期的指示”。针对这三种“训练”的商业市场正不断扩大。经历过这些困难的机构给出的意见非常统一：购买现成的，不要试图构建自己的。购买库、培训、工具和咨询。如果一个公司试图提供自足的工具，同时又制作应用程序或系统，那么它将发现很难取得成功。

很少人会认为编码规范是“理想的”，甚至都算不上“良好”。但在上述的机构中，编码规范又是必要的。

以下条目给出了一些基本的约定和风格方面的指南。

27.3 我们机构应该根据以前用C的经验来制定编码规范么？

不要！

不管你用C的经验多么丰富，不管你对C掌握的多么熟练，一名好的C程序员不一定就会是一名好的C++程序员。从C转换到C++，并不只是学习一下C++中++部分的语法和语义。那些希望获得OO好处的机构，如果不在“OO编程”中真正运用“OO”技术，那么他们就是在自欺欺人；他们的蠢行最终会在财报上表现出来。

应该由C++专家来打造C++的编码规范。开始可以在comp.lang.c++上问问题。寻找能够帮助你避开陷阱的专家。购买程序库，然后看“好”的库能否通过你的编码规范。在获得足够的C++经验之前，不要自己制定编码规范。没有标准要好过一份糟糕的标准，因为不合适的“官方”标准会让错误的做法一直存在。现在C++培训和程序库的市场正不断壮大，可以从中吸取经验。

还有：只要对某件事物有需求，就会增加出现伪专家的机会。要三思而后行。同时还要从过去的公司中寻求反馈，因为有娴熟技术的人未必是一名好的沟通者。最后，选一名会教学生的专业人，注意这可不是有足够语言/编程范式相关知识的全职教师。

27.4 `<xxx>` 和 `<xxx.h>` 这两种头文件有何不同？

ISO C++标准的头文件不包含 `.h` 后缀。标准委员会修改了以前的做法。C的头文件和C++头文件在细节上不同。

C++标准库保证包含来自C语言的那18个标准头文件。这些头文件有两种标准风格：`<cxxx>` 和 `<xxx.h>` （这里 `xxx` 是头文件的基本文件名，例如 `stdio` ， `stdlib` 等等）。这两种风格的头文件是一样的，但有一点不同： `<cxxx>` 风格的头文件将所有声明放在 `std` 名字空间中，而 `<xxx.h>` 除了将声明放在 `std` 名字空间，同时还放在了全局名字空间。委员会这么做是为了已有的C代码能够被C++编译器编译。但 `<xxx.h>` 是已经过时了，虽然现在仍被标准接受，但可能在以后的标准中不再支持。（参见ISO C++标准的D.5节）。

C++标准库还增加了32个C里面没有直接对应的标准头文件，例如 `<iostream>` , `<string>` 和 `<new>` 。你可能会在老的代码中看到#include `<iostream.h>` 这种写法，因此一些编译器厂商还提供这些 `.h` 版本的头文件。但要注意， `.h` 版本的头文件可能与标准版本不一样。如果一个程序里有些用 `<iostream>` ，有些用 `<iostream.h>` ，那这个程序可能无法正常运行。

在新项目中，应当用 `<xxx>` ，而不该用 `<xxx.h>` 。

当修改或扩展使用旧式头文件名的代码时，最好还是遵从那些代码的做法，除非有很重要的理由换用标准头文件（例如标准 `<iostream>` 提供了一些功能，而厂商的 `<iostream.h>` 里没有）。如果想要使以后代码符合标准，那么要确保在所有被链接起来的代码中包括外部库，里面所用的所有C++头文件都修改了。

以上内容只对标准头文件有影响。你自己的头文件可以随便怎么命名，参见[27.9]。

27.5 我应该在我的代码中使用 `using namespace std` 么？

可能不该。

人们不喜欢一边又一遍地键入 `std::` 。他们发现 `using namespace std` 能够使编译器看到任何 `std` 中的名字，即使名字前没有被 `std` 限定也能看到。问题是这会使编译器看到所有在 `std` 中的名字，包括那些你没想到的名字。换句话说，这可能导致名字冲突和二义性。

例如，假设你的代码需要计数，然后你定义了一个名为 `count` 的变量或函数。但 `std` 库也使用 `count` 这个名字（这是一个 `std` 算法），这就可能导致二义性。

你看，名字空间的用处就是用来防止两部分独立开发的代码产生名字冲突。 `using` 指令（描述 `using namespace XYZ` 的术语）实际上是把一个名字空间的内容全部引入到另外一个名字空间了，这就违背了名字空间的本意。 `using` 指令是为了使遗留的C++代码容易迁移到名字空间上来，但至少在新的C++代码中，不该大范围这么做。

如果真的不想敲 `std::`，可以使用 `using` 声明，或使自己适应 `std::`（不是办法的办法）

- 使用 **`using`** 声明，这可以引入特定的名字。例如，为了能够在代码中使用 `count` 同时还不必写 `std::`，可以在代码中插入一行 `using std::count`。这不大可能会带来混乱或二义性，因为是显式引入名字的。

```
#include <vector>
#include <iostream>

void f(const std::vector<double>& v)
{
  using std::cout;  // ← using声明允许直接使用cout，前面不必限定。

  cout << "Values:";
  for (std::vector<double>::const_iterator p = v.begin(); p != v.end(); ++p)
    cout << ' ' << *p;
  cout << '\n';
}
```

- 让自己适应 **`std::`**（不是办法的办法）：

```
#include <vector>
#include <iostream>

void f(const std::vector<double>& v)
{
  std::cout << "Values:";
  for (std::vector<double>::const_iterator p = v.begin(); p != v.end(); ++p)
    std::cout << ' ' << *p;
  std::cout << '\n';
}
```

我个人觉得与其为每个不同的 `std` 名字决定是否使用 `using` 声明、并且找到最适合放置这个声明的地方，不如直接敲"`std::`"，这样还更快。但这两种方法都不错。记住你是一个团队的一分子，所以要确保使用的方法和其它人保持一致。

27.6 `?:` 操作符能够写出难以阅读的代码，它是邪恶的么？

不是。但和往常一样，记住可读性是最重要的事情之一。

有人觉得应避免使用 `?:` 运算符，因为和 `if` 语句相比，它有时会令人困惑。在很多情况下，`?:` 常会使代码更难读懂（因此应该替换为 `if` 语句）。但有时用 `?:` 更清晰，因为这会强调到底在干什么事情，而不是强调那里有个 `if`。

让我们先来看一个很简单的例子。假设你需要打印出一个函数调用的结果。这时你应该把真正的目的（打印结果）放在开头，然后把函数调用放在后面，因为函数调用是相对次要的（直觉上大多数开发者认为一行开头的内容是最重要的）。

```
// 更好（强调主要目的—打印）：
std::cout << funct();

// 不那么好（强调次要目的—函数调用）：
functAndPrintOn(std::cout);
```

现在我们把这个观点扩展到 `?:` 上来。假设你的真正目的是要打印一些东西，但需要做一些额外操作来决定打印的内容。因为在概念上打印是更重要的事，所以我们倾向于把它放在开头，而把用于判断的逻辑放在后面。在下面的例子中，变量 `n` 代表消息发送者的数量；消息本身被打印到 `std::cout` 上：

```
int n = /*...*/;   // 发送者的数量
// 更好（强调主要目的—打印）：
std::cout << "Please get back to " << (n==1 ? "me" : "us") << " soon!\n";

// 不那么好（强调次要目的—函数调用）：
std::cout << "Please get back to ";
if (n == 1)
  std::cout << "me";
else
  std::cout << "us";
std::cout << " soon!\n";
```

已经说过了，通过不同组合使用 `?:` 、 `&&` 和 `||` 等运算符，可以写出很过分且难以阅读的代码（“只写代码”）。例如

```
// 更好（意思很明显）：
if (f())
  g();

// 不太好（更难理解）：
f() && g();
```

我个人觉得这里明确写出 `if` 会更清晰，因为这强调主要的事情（至于要做什么是根据 `f()` 的结果来决定的），而不是强调次要的事情（调用 `f()` ）。换句话说，在这里使用 `if` 是恰当的，原因和上面用 `if` 是不恰当的一样：我们希望把主要事情放在在显眼位置，次要事情放到次要位置。

不管怎样，别忘了可读性是最终目的（至少是目的之一）。你的目标不应是为了避免类似 `?:` 、 `||` 、 `if` 或者甚至是 `goto` 这样的语法结构。如果你变成一个“唯标准论者”，那么你最终会另自己蒙羞，因为任何基于语法的规则总是存在反例。如果你是强调大的目标和指导原则（例如“主要的事情要放在显眼位置”，或者“把重要事情放在开头”，甚至是“让你的代码含义明显容易阅读”），那就好多了。

写出来的代码是要被其它人读的，不是给编译器看的。

27.7 我应该把变量声明放在函数体中间还是开头？

在第一次使用的附近声明。

对象是在声明时被初始化（构造）的。如果在函数中间才有足够的信息来初始化一个对象，那就应该把声明放在中间，以便对象可以正确初始化。不要现在开头给对象初始化为一个“空值”，然后在其它地方给它实际“赋值”。这么做是为了运行时的效率。与其先把对象构造到一个错误状态，然后再修正，不如开始就把对象构造正确，这样速度更快。简单的例子表明对像 `String` 这样简单的类，也会有350%的速度差别。具体的数值可能有所不同，而且整个系统的效率损失肯定小于350%，但的确会有效率损失。不必要的效率损失。

对这个问题一个常见的回应是：“我们要为对象中的每个数据提供一个 `set()` 成员函数，这样构造对象的代价就被平摊开了。”这就不仅是损失效率了，因为还导致维护困难。为每个数据成员提供 `set()` 函数和 `public` 数据一样糟糕：你把实现技术暴露给外部了。你唯一隐藏掉的是成员对象的物理名字，而具体的实现细节（比方说用了一个 `List` 、一个 `String` 和一个 `float` ），则还是给外面知道了。

底线是：局部变量应在靠近第一次使用的地方声明。对C语言专家来说可能不太习惯，但新事物不一定就不好。

27.8 哪种源代码文件名最好？ `foo.cpp` ？ `foo.C` ？ `foo.cc` ？

如果已经有一种命名约定了，那就继续使用。否则，需要查一下所使用的编译器接受哪种文件名。常用的是： `.cpp` ， `.C` ， `.cc` 或 `.cxx` （当然如果用 `.C` 的话，那么文件系统需要能够区分大小写，以便不会混淆 `.C` 和 `.c` ）。

我们已经使用 `.cpp` 做为C++源文件名的后缀了，我们也用 `.C` 。如果用 `.C` ，那么在把代码移植到大小写不敏感的文件系统时，需要告诉编译器将 `.c` 文件做为C++源文件对待（例如IBM CSet++是用 `-Tdp` 选项，Zortech C++编译器用 `-cpp` ，Borland C++编译器用 `-P` ）。

关键在于这些文件扩展名中，并不存在说哪个比其它更好。我们通常根据客户的要求来选择（这些问题应当依据商业上的考量，而不是技术）。

27.9 哪种头文件名最好？ `foo.H` ？ `foo.hh` ？ `foo.hpp` ？

如果已经有一种命名约定了，那就继续使用。如果还没有，而且不需要让编辑器区分C和C++文件，那就用 `.h` 好了。否则，就按编辑器的要求来，例如 `.H` 、 `lhh` 或 `.hpp` 。

我们倾向于使用 `.h` 或 `.hpp` 做为C++头文件的后缀名。

27.10 C++有没有一些像 `lint` 一样的规范原则？

有的。有些做法一般被认为是危险的。但没有一个是总是“不好”的，因为最糟糕的做法有时也有用武之地。

- `class Fred` 的赋值运算符应该将 `*this` 做为 `Fred&` 返回（允许将赋值运算串起来）
- 一个类如果有虚。
- 一个类如果有{析构函数、赋值运算符、拷贝构造函数}中的任何一个，一般也需要另外两个。
- `class Fred` 的拷贝构造函数和赋值运算符的参数应该用 `const` 来限定，即 `Fred::Fred(const Fred&)` 和 `Fred& Fred::operator=(const Fred&)`
- 当在构造函数中初始化对象成员时，总是使用初始化列表，而不是用赋值。对于用户定义的类来说，这两种办法在性能上可能会有很大差别（3倍！）
- 赋值运算符需要保证当对自身赋值时不做任何操作，否则可能会有麻烦。有时这要求做显式的判断。
- 重载运算符时，要遵守指导原则。例如，如果类里面重载了 `+=` 和 `+` ，那么 `a += b` 和 `a = a + b` 一般来说应该是做相同的操作。其它内建/基本的类型也是如此（例如 `a += 1` 和 `++a` ； `p[i]` 和 `*(p+i);` 等等）。在编写二元运算符时，可以使用 `op=` 这种形式来强制达到这个目的。例如：

  ```
  Fred operator+ (const Fred& a, const Fred& b)
  {
    Fred ans = a;
    ans += b;
    return ans;
  }
  ```

  用这种办法，那些“构造性的”译注1二元运算符就不必成为类的友元了。但有时可以更高效地实现一些普通的操作（例如，如果 `class Fred` 是 `std::string` 类型，并且 `+=` 需要重新分配/拷贝字符串内存，那么最好在开始就能够知道最终的长度）。

27.11 为何人们对指针转换和/或引用转换如此担忧？

因为它们是邪恶的！（这说明在使用它们时需要很小心谨慎）。

不知为什么，程序员在转换指针时不太注意。他们到处转换指针类型，然后还奇怪为什么会出问题。最糟糕的是，当编译器给出一条错误消息时，他们就添加一个类型转换“让编译器闭嘴”，然后他们再“测试一下”看能否运行。如果你做了很多指针或引用的类型转换，请继续往下读。

当你转换指针类型和/或引用类型时，编译器通常不会产生任何信息。指针类型转换（和引用类型转换）会使编译器保持沉默。我把他们当作是一种错误信息的过滤器：编译器想要抱怨，因为它发现你正在做蠢事，但同时也发现它不该抱怨因为你用了类型转换，所以编译器就把错误消息丢掉了。这就像用密封胶带封住编译器的嘴：它试图告诉你一些重要事情，而你却故意让它闭嘴。

指针类型转换告诉编译器："别想了，赶紧生成代码；我很聪明，你太笨了；我很伟大，你很渺小；我知道我在做什么，所以就假装这是汇编语言，然后生成代码吧。"当你转换类型时，编译器就盲目地生成代码—由你来控制（和负责）生成的结果。编译器和语言会缩减（甚至是消除）你所能得到的保证。你只能靠自己了。

做个类比，虽然手抛链锯玩完全合法，但这么做却很蠢。如果出了问题，别向链锯制造商抱怨—你做了他们没有保证的事情。你只能靠自己。

为公平起见，语言的确在类型转换时做了一些保证，至少是在一个有限的子集内有保证。例如，语言保证当从对象指针（指向一块数据的指针，不是指向函数，也不是指向成员）转换到void*，并且再转换原数据类型时，是没有问题的。但很多时候，你只能靠自己。）

27.12这两种标识符的名字：`that_look_like_this` 和 `thatLookLikeThis` ，哪种更好？

这个要看以前是怎么做的。如果你有Pascal或Smalltalk背景，那么会喜欢 `youProbablySquashNamesTogether` 。如果有Ada背景，那么会喜欢 `You_Probably_Use_A_Large_Number_Of_Underscores` . 如果有微软Windows背景，那么可能倾向于"匈牙利"命名法，即在标识符前面添加表示类型的前缀译注1。对于Unix C背景的人来说，会喜欢用缩写[译注2]。

所以没有普遍适用的标准。如果你在的项目团队已经有一份命名规范了，就照上面说的做。如果硬要推翻重来，可能更多会带来争吵而不是解决问题。从商业角度来看，只有两件事是重要的：代码可读性好，团队中的每个成员都使用相同风格。

除此之外，差别很小。

还有，在使用平台相关的代码时，不要用一种完全不同的风格。例如，一种编码风格在使用微软的库时可能看起来很自然，但在和UNIX库一起使用时就会看起来很奇异。别这么做。为不同的平台使用不同的风格。（为避免有人不仔细看，别给我发email询问那些要移植到（或是用在）不同平台上的通用代码，因为这些代码不是平台相关的，所以刚才说的"为不同的平台使用不同的风格"在这里并不适用。）

好吧，还有。真的。别跟自动生成的代码（例如通过工具产生的代码）过不去。一些人对编码规范抱有一种宗教般的狂热，他们试图让工具产生的代码符合他们的风格。别这么做，即使工具产生的代码风格不同，也别管它。记住钱和时间才重要？！？整个编码规范目的就是为了省钱省时间。别把这个变成烧钱的陷阱。

译注1：原文是jkuidsPrefix vndskaIdentifiers ncqWith ksldjfTheir nmdsadType

[译注2]: 原文是abbr evthng n use vry srt idntfr nms. (AND THE FORTRN PRGMRS LIMIT EVRYTH TO SIX LETTRS.)

27.13从哪里可以找到一些编码规范么？

有好几个地方可以找到。

在我看来，Sutter和Alexandrescu的"C++ Coding Standards"（220页，Addison-Wesley出版，2005, ISBN 0-321-11358-6）是最好的。 我有理由推荐此书，并且本书作者很能激发推荐者的热情。所有人都大力推荐，以前我可没见过这事。

这里有一些编码规范，可以以此为起点来制定机构的编码规范。（列表顺序是随机的）（有些已经过时了，有些可能非常糟糕。我不会推荐任何一种。使用者自己注意。）

- `www.codingstandard.com/`
- `cdfsga.fnal.gov/computing/coding_guidelines/CodingGuidelines.html`
- `www.nfra.nl/~seg/cppStdDoc.html`
- `www.cs.umd.edu/users/cml/resources/cstyle`
- `www.cs.rice.edu/~dwallach/CPlusPlusStyle.html`
- `cpptips.hyperformix.com/conventions/cppconventions_1.html`
- `www.objectmentor.com/resources/articles/naming.htm`
- `www.arcticlabs.com/codingstandards/`
- `www.possibility.com/cpp/CppCodingStandard.html`
- `www.cs.umd.edu/users/cml/cstyle/Wildfire-C++Style.html`
- Industrial Strength C++
- Ellemtel的编码规范在这里可以找到：
    - `membres.lycos.fr/pierret/cpp2.htm`
    - `www.cs.umd.edu/users/cml/cstyle/Ellemtel-rules.html`
    - `www.doc.ic.ac.uk/lab/cplus/c++.rules/`
    - `www.mgl.co.uk/people/kirit/cpprules.html`

注意：

- Ellement的标准已经过时了，但鉴于其重要地位，所以仍然列出。它是第一个广泛传播并被采用的C++编码规范，也是第一个批判使用保护成员的。
- Industrial Strength的C++规范也过时了，但在那些提到在基类中使用保护非虚析构函数的规范中，它是第一个被广泛发行的。

27.14 我应该用“不常见”的语法么？

只有当有足够的理由时再去用。换句话说，就是通过“普通”的语法无法获得同样的结果。

决定软件方面决策的是钱。除非你是在象牙塔中，否则，当你的做法会增加费用、增加风险、增加时间，或者是在一个受限环境中增加产品的时空开销，那么你的做法就不好。在意识中，你应该把这些都转换为钞票。

根据这种以实用为目的、以钞票为导向的观点，只要有等价的“正常”语法，程序员就应避免实用非主流的语法。如果一个程序员写下隐晦的代码，其它程序员看了会困惑，这就会耗费金钱。其它程序员可能会引入bug（会花钱），可能会需要更长的时间来维护（钱），修改起来可能会很困难（错过了市场机遇等于损失了钱），可能在优化时更困难（在受限的环境中，有人会需要为更大内存、更快CPU和/或更大电池来买单），另外客户可能还不满意（钱）。这是一种有关风险和回报的权衡。但如果有等价的“正常”语法能够达到同样目的，那么再努力降低使用“非正常”语法所带来的风险，就没有任何“回报”。

例如，在混乱C代码大赛中使用的技术，礼貌来讲是不正常的。没错，其中很多是合法的，但不是所有合法的事情都合理。使用奇怪的技巧会使其它程序员感到困惑。一些程序员喜欢“秀”他们挑战极限的能力，但这是把自我的虚荣心放在了比钱更重要的位置，是不专业的表现。坦白说，任何这么干的人都该被开除。（如果你觉得我太“刻薄”或“残忍”，我建议你调整一下态度。记住：公司雇你来是为了来帮助它而不是来伤害它的。那些把自我放到公司最佳利益上的人应该被开除出去）。

举个非主流语法的例子，`?:` 运算符一般不作为语句来用。（一些人甚至不喜欢把它用在表达式里。但必须承认有很多地方用到了 `?:` ，所以不管喜欢不喜欢，（用作表达式）是“正常”的。这里有个把?:用作语句的例子：

```
blah();
blah();
xyz() ? foo() : bar();  // 应该用if/else
blah();
blah();
```

还有把 `||` 和 `&&` 当作"if-not"和"if"语句来用也是一样道理。是的，Perl里面有这些惯用法，但C++不是Perl，用这些来代替 `if` 语句（而不是用在表达式中）在C++中是“不正常”的。例如：

```
foo() || bar();  // 应该用if (!foo()) bar();
foo() && bar();  // 应该用if (foo()) bar();
```

这里还有个例子，好像是能够运行，甚至是合法的，但绝不是正常的。

```
void f(const& MyClass x)  // 应该用const MyClass& x
{
  ...
}
```

[28] 学习OO/C++

FAQs in section [28]:

28.1 什么是师徒指导？

这是学习OO最有效的办法。

用面向对象的思考方式经过努力学来的，不是光靠老师教就可以的。跟那些眞正知道自己在说些什么的人混熟，研究他们的思考方法，观察他们是如何解决问题的。倾听他们的言论。通过模仿来学习。

如果你在一家公司工作，那么让公司为你派一个指导者。我们见过有公司浪费了很多钱，这些公司希望能够“省钱”，于是就仅仅为雇员买几本书（“书放在这里了，周末读一遍；到礼拜一，你就学会OO了”）。

28.2 在学习OO/C++之前我应该先学C吗？

不用费那个劲。

如果你最终的目标是学习OO/C++并且还不会C，那么读有关C的书籍和参加学习C的课程只会浪费你的时间，而且还会教你一堆在你以后学OO/C++时要忘掉的东西（例如malloc(), 不必要的switch语句，等等）。

如果你想学OO/C++，那就直接学这个。另外花时间学C只会浪费你的时间，还会迷惑你。

28.3 在学习OO/C++之前我应该先学Smalltalk吗？

不用费那个劲。

如果你最终的目标是学习OO/C++并且还不会Smalltalk，那么读有关Smalltalk的书籍和参加学习Smalltalk的课程只会浪费你的时间，而且还会教你一堆在你以后学OO/C++时要忘掉的东西（例如动态类型，非子类化的继承（non-subtyping inheritance），用错误码处理异常，等等）。

如果你想学OO/C++，那就直接学这个。另外花时间学Smalltalk只会浪费你的时间，还会迷惑你。

注意：我是ANSI C++(X3J16)标准委员会的成员。我不是什么语言的死忠。我没说C++和Smalltalk哪个好哪个坏。我只是说它们是不同的语言。

28.4 我只买一本书就够了么？还是需要买几本？

至少3本。

在用C++进行OO编程的领域里，有3类知识需要学习。应该在每一类里都买一本好书，而不应买一本还凑活的书。这3类包括：

- 合法C++的指南——在C++里哪些能做，哪些不能做。
- 合理C++的指南——在C++里哪些应该做，哪些不应该做。
- 通过例子讲解编程的指南——演示很多例子，这通常会大量使用C++标准库

合法性指南会按一种平等的方式讲解所有的语言特性。合理性指南专注于那些在通常的编程任务中你应该使用的方法。合法性指南教会你如何让程序通过编译器的检查。合理性指南则指导你何时使用或放弃一项语言特性。

注

- 不要在这几个类别之间权衡。不应重视只一个而忽略其它。它们要配合起来才有用。
- “合法性”与“合理性”都是必要的。你应该把它们都掌握好。

除了这些（强调“附加事项”），你应该考虑在其它两个领域内至少各买一本书：至少一本有关OO设计的和至少一本编码标准的。讲设计的书训练你在更高层次上用对象来思考问题，编码标准则可以为你所在的机构推广最佳实践，还能帮助人们容易读懂别人写的代码（例如如果某个团队落后了，你可以调人上去。）

28.5 有哪些讲合理使用C++的好书？

这里有些我个人（经过仔细筛选的）认为必读的书，按作者姓名的字母顺序排列：

- Cline, Lomow, and Girou, C++ FAQs, Second Edition, 587 pgs, Addison-Wesley, 1999, ISBN 0-201-30983-1. 以类似FAQ一问一答的形式覆盖了大约500个方面的话题。
- Meyers, Effective C++, Second Edition, 224 pgs, Addison-Wesley, 1998, ISBN 0-201-92488-9. 以短文的形式探讨了50个话题。

- Meyers, More Effective C++, 336 pgs, Addison-Wesley, 1996, ISBN 0-201-63371-X. 以短文的形式探讨了35个话题。

相似点：这几本书都给出了很多代码示例。都是非常优秀、有见地、有用的好书。都有很好的销量。

_不同点：_Cline/Lomow/Girou书中的示例都是完整可运行的，不是代码片段或单独的类。Meyers的书用了很多图例来说明问题。

28.6 有哪些讲合法使用C++的好书？

这里有些我个人（经过仔细筛选的）认为必读的书，按作者姓名的字母顺序排列：

- Lippman, Lajoie and Moo, C++ Primer, Fourth Edition, 885 pgs, Addison-Wesley, 2005, ISBN 0-201-72184-1. 可读性很好
- Stroustrup, The C++ Programming Language, Third Edition, 911 pgs, Addison-Wesley, 1998, ISBN 0-201-88954-4. 包含了很多内容

相似点：这两本书都很好地概括了几乎所有的语言特性。我在连续两期C++ Report上分别评论了这两本书。我评论说这两者都是顶尖的好书。都有很好的销量。

不同点：如果你不懂C，那么Lippman等人的书比较适合。如果你了解C并且向快速了解很多东西，Stroustrup的书更合适。

28.7 有哪些通过例子讲解C++编程的好书？

这里有些我个人（经过仔细筛选的）认为必读的书，按作者姓名的字母顺序排列：

- Koenig and Moo, Accelerated C++, 336 pgs, Addison-Wesley, 2000, ISBN 0-201-70353-X. 很多使用C++标准库的例子。真正是一本通过例子讲解编程的书
- Musser and Saini, STL Tutorial and Reference Guide, Second Edition, Addison-Wesley, 2001, ISBN 0-201-037923-6. 用很多例子说明如何使用C++标准库的STL部分，还有很多基本的小细节。

28.8 还有哪些与OO/C++相关的讲OO的书？

有的！很多！

上面列出的合理性、合法性和例子讲解的几类书都是和OO编程相关的。在有关OO分析与设计的领域中，也有很多好书。

在这些领域中有大量的好书。我个人（经过深思熟虑）认为，在OO设计模式方面最重要的必读书是：Gamma et al., Design Patterns, 395 pgs, Addison-Wesley, 1995, ISBN 0-201-63361-2. 此书描述了在好的OO设计中常会出现的“模式”。如果你准备做OO设计工作，那就一定要读这本书。

[31] 引用与值的语义

FAQs in section [31]:

31.1 什么是值和/或引用传递，在C++用哪个最好？

对于引用，被赋值的是一个指针拷贝。而值传递，被赋值的是值的拷贝而不是指针。C++中你可以选择使用赋值操作符或者拷贝值（值传递），或者使用指针拷贝来复制一个指针（引用传递）。C++也允许你重写复制操作符来实现你想要的操作，但是默认选择是拷贝值。

引用传递的好处：灵活和动态绑定（只有使用指针或者引用的时候，才能获得动态绑定）。

值传递的好处：速度。因为值传递需要拷贝一个对象（而不是一个指针），你可能很奇怪为什么会这样。事实是大家通常使用一个对象，而不是拷贝多个对象，因此偶尔的拷贝开销比间接指针访问对象带来的开销要小。

三种情况你会获得一个对象而不是对象指针：本地对象，全局或者静态对象，以及类的非指针成员对象。最重要的是后者（对象组合）。

下个FAQ会给出更多的值/引用传递的信息。请阅读所有内容以有个全面认识。前几个倾向于使用值传递，如果你只阅读前几个，可能你会得到一个片面的认识。

赋值还包括其他问题（比如浅拷贝和深拷贝），这里不讨论这些。

31.2 什么是“虚成员”，如何/为什么在C++中使用？

"虚成员(Virtual Data)"允许子类改变父类的成员对象。C++并不严格支持“虚成员”，但是可以模拟实现。虽然实现的不是很漂亮。

模拟实现要求基类要有一个成员对象指针，子类必须提供一个新对象，基类的成员对象指针指向这个新对象。基类可以有一个或者多个正常的构造函数提供成员指针对象的对象（通过 `new` ），基类的析构函数将会“ `delete` ”这个对象。

例如， `Stack` 类可能有个 `Array` 成员对象（使用指针）而子类 `StrechableStack` 可以重写基类的 `Array` 成员为 `StrechableArray` 。要使这个实现， `StretchableArray` 必须从 `Array` 继承，这样 `Stack` 类可以使用 `Array*` 。 `Stack` 类的正常构造函数可以初始化 `Array*` 为 `new Array` , 但是 `Stack` 类也要有一个构造函数（很可能 `protected` 属性的构造函数)可以接受一个来自子类的 `Array*` 。 `StretchableStack` 类的构造函数为基类的这个特殊构造函数提供 `new StretchableArray` 对象。

好处:

- 易于实现 `StretchableStack` (多数代码可以被继承)
- 用户可以传递 `StrechableStack` 为 `Stack` 类型的参数或者变量

缺点:

- 为访问 `Array` 增加了额外层
- 为堆内存分配需要增加额的 `new` 和 `delete` 操作
- 增加了额外的动态绑定开销 (理由见下节FAQ)

换句话讲，我们简化了 `StrechableStack` 的实现代码，但是所有的用户都要付出代价。不幸的是，不仅 `StrechableStack` 用户而且 `Stack` 用户都要付出这个代价。

请阅读本节其他内容。（这样你会有一个全面认识）

31.3 怎么区别虚拟数据和动态数据？

最简单的办法是虚函数分析法。虚函数: 虚函数意味着“声明（签名）”在子类中必须一样，但是“定义（实现）”可以被重写。继承的成员函数的重写是子类的静态属性，不会随着任何特定对象的改变而动态改变，也不可能应为子类的不同实例而有不同的实现。

现在重新阅读上面段落，但是要做下面替换：

- "成员函数" → "成员对象"
- "签名" → "类型"
- "实现" → "确切类"

这样你就可以定义“虚数据”。

另外一种方法是辨别"per-object"成员函数和"dynamic" 成员函数。 "per-object" 成员函数是指在不同的实例中实现有可能不同的成员函数，可以使用函数指针实现，这个指针可以是 `const` ，因为该指针在对象的生命周期中不会被改变。而"dynamic" 成员函数是指将会随时间而动态改变的成员函数，也可以由函数指针实现，但是函数指针不能为 `const` 。

概括一下上面的分析，数据成员有三种概念：

- 虚数据: 类的成员对象定义可以在子类中被重写，假设成员对象的生命（类型）相同。这种重写是子类的静态属性。
- per-object-data: 任何类的既定对象可以在初始化（wrapper对象）的时候实例化一个不同conformal（相同类型）的成员对象，成员对象的确切类是Wrapper类的静态属性。
- dynamic-data: 成员对象的确切类可以被动态改变。

他们相似的原因是他们都不被C++支持，只有很少情况下可以这样使用。在这种情况下，模拟机制都是相同的：通过指向基类（很可能是抽象类）的指针。在支持“first class” abstraction mechanisms的语言中，可能这种区别很明显一些，因为他们将会有各自不同的语法表示。

31.4 应该通常使用数据成员对象指针或者使用“组合”？

组合。

一般来说，你的成员对象应该被包含在组合对象中（并不总是这样，包装器（Wrapper）对象是一个你可以使用指针或者引用的好例子; 而N-to-1-uses-a关系也需要指针或者引用）。

完全包含成员对象性能优于指针的原因有三点：

- 访问对象时候是否需要额外的间接访问
- 额外的堆内存分配(在构造函数中使用 `new` ，在析构函数中使用 `delete`)
- 额外的动态绑定(理由见下面FAQ)

31.5 什么是使用成员对象指针的3个相对性能开销？

前一节FAQ列举了3个相对性能开销: :

- 就自身来说，一个额外的间接访问开销不值一提。
- 堆内存分配可能成为一个性能问题（ `malloc` 的传统实现的性能会下降，随着内存分配的增加; 面向对象软件很容易使得内存分配增加，除非你很细心）。
- 额外的动态绑定来自于对象指针，而不是对象。只要C++编译器能够知道确切的 `class` ， 虚函数调用就会被静态绑定，静态绑定允许内联。而内联将会带来成千上万的优化机会，比如 procedural integration, register lifetime issues等等。下面三种情况下C++编译器能够知道对象确切的 `class` ： 本地变量，全局/静态变量，完全包含的成员对象。

因此完全包含成员对象允许重要的优化，而这在使用对象指针的情况下是不可能的。这是具有引用语义的编程语言为什么面临继承性能挑战的主要原因。

请阅读下面***3***个***FAQ***一遍获得全面理解！

31.6 “内联虚函数”的会被“内联”吗？

有时...

当对象是个指针或者引用的时候， 虚函数调用不能被内联，因为函数必须被动态调用。原因：编译器无法知道实际的代码来调用直到运行时（即动态），因为该代码可能是来自一个派生类，调用函数编译以后才创建的。

因此，只有当编译器知道虚函数调用的目标的“确切类”的时候，”内联虚函数”才有可能被内联。发生这种情况只有在编译器知道一个实际的对象,也就是说，本地对象，全局/静态对象，或在组合的完全包含对象，而不是一个指针或引用的时候。

注意，内联和非内联之间的差别远远超过普通函数调用和虚函数调用的差别。例如，普通函数调用和虚函数调用的差别常常只有两个额外的内存引用，但内联函数和非内联函数的差别可以多达一个数量级（数以亿计的调用无关紧要的成员函数， 内联虚函数的损失可能会导致25倍的差距！Doug Lea, "Customization in C++," proc Usenix C++ 1990。

这种顿悟的实际后果：不要陷在无休止的辩论中（或销售策略！），来比较编译器/语言的虚函数调用的成本。和具有扩展“内联”成员函数调用的语言/编译器做比较是没有任何意义的。也就是说，许多语言实现厂商带鼓吹他们的调度策略是如何好，但如果没有内联成员函数的话，系统的整体性能会很差，正是因为靠内联调度，他们才具有最好的性能。

注意：请阅读下面的***2***个***FAQs***一遍了解另一方面！

31.7 听起来像我不应该使用引用？

不对。

引用是个好东西。我们不能生活在没有引用。我们只是不希望我们的软件使用太多的指针。在C++中，你可以挑选你想要引用语义（指针/引用）以及值语义（如对象包含其他对象等）。在一个大的系统中，应该有一个平衡。然而，如果你无论什么都使用指针的话，你将得到许多速度方面的问题。

求解问题的对象往往要比更高层次的对象占用更多的存储空间。这些”问题空间“抽象类的ID通常比他们的“值”更重要，因此以用语义应该被用于求解问题的对象。

请注意，这些求解问题的对象通常在较高的抽象层次，相比那些处于解决方案空间的对象来说。因此求解问题的对象通常有一个相对较低的使用频率。因此，C ++中为我们提供了一个理想的情况：对于那些需要独特的身份的对象，或过大而不能复制我们选择使用引用语义，对于其他对象我们可以选择值语义。因此，最高使用频率的对象将最终使用值语义，因为在灵活性方面我们没有损失，但是在性能方面，实现了我们最需要的！

这些只是真正的面向对象设计的诸多问题中的一部分。精通面向对象设计/C++需要需要时间和高质量的训练。如果你想有一个强有力的工具，你要投入时间和精力。

不要停下来！__无比阅读下一个问题！！

31.8 引用的性能问题是否意味着我要使用值传递？

不是。

前面FAQ谈论的是成员对象，而不是参数。一般而言，对象是继承层次结构的一部分，应该通过引用或指针来传递，而不是值传递，因为只有这样你才能得到（期望的）动态绑定（按值传递和继承不相符，因为派生类对象将会被切片 ，当按值传递到一个基类对象的时候）。

除非另有其他的理由，成员对象应当按值传递，参数应按引用传递。以前的FAQ里面讨论了应该按引用传递的成员对象的“其他的理由”。

[32] 如何混合C和C++编程

FAQs in section [32]:

32.1 混合C和C++编程时我需要知道什么？

以下是一些要点（虽然有些编译器供应商可能不需要全部要点，查看你的编译器供应商的文档）：

- 你必须使用你的C++编译器来编译的main()（也就是说静态初始化）
- 你的C++编译器应该能够进行直接链接（也就是说，它有自己的专门类库）
- 你的C和C++编译器可能需要来自同一个供应商，并具有兼容的版本（也就是说，它们有相同的调用约定）

此外，您需要阅读本节的其余部分，了解如何使你的C可调用的C++函数和/或C++可调用C函数。

顺便说一下，还有另一种途径来处理这件事：使用C++编译器编译所有代码（甚至是你的C风格代码）。这几乎无需混合C和C++，但是你要格外小心（也可能是，希望！ -发现了一些错误）你的C风格的代码。缺点是你需要更新你的C代码风格，主要是因为C++编译器比C编译器更加严谨/挑剔。值得一提的是，清理你的C代码风格可能会比实际混合C和C++所付出的努力要少，并且清理C代码风格还能给你带来一份额外收入。但是很明显，你几乎没有选择的余地，如果你不能改变C代码（例如，如果它是来自第三方的）。

32.2 如何在C++代码中包含标准的C头文件？

要包含一个标准（如 `<cstdio>` ）头文件，你不需要做任何事情。例如：

```
// This is C++ code

#include <cstdio>                // Nothing unusual in #include line

int main()
{
  std::printf("Hello world\n");  // Nothing unusual in the call either
  ...
}
```

如果你认为 `std::printf` 的 `std` 部分很奇怪，那么最好的办法是“适应它”。换句话说，它是使用标准库函数的标准方法，所以你不妨现在开始习惯它。

然而，如果你正在使用你的C++编译器编译C代码，你恐怕不想修改所有这些 `printf` 调用为 `std：：printf` 。幸运的是，这种情况下，C代码将使用旧式头 `<stdio.h>` 而不是新型头 `<cstdio>` ，命名空间技术将会照顾一切：

```
/* This is C code that I'm compiling using a C++ compiler */

#include <stdio.h>          /* Nothing unusual in #include line */

int main()
{
  printf("Hello world\n");  /* Nothing unusual in the call either */
  ...
}
```

最后的评论：如果你有不属于标准库C头文件，你需要遵守一些不同的准则。有两种情况：要么你不能改变头文件，要么你可以改变头文件。

32.3 如何在C++代码中包含非系统的C头文件？

如果你是其中一个C头文件不是由系统提供的，你可能需要把 `#include` 行放到 `extern"C"（/ * ... * /）` 构造中。这告诉C++编译器的功能在头文件中声明的C函数。

```
// This is C++ code

extern "C" {
  // Get declaration for f(int i, char c, float x)
  #include "my-C-code.h"
}

int main()
{
  f(7, 'x', 3.14);   // Note: nothing unusual in the call
  ...
}
```

注： 对于系统提供的C头文件（如 `<cstdio>` ）和你可以更改的C头文件，准则略有不同

32.4 如何修改我自己的C头文件，以便更容易的在C++代码中包含他们？

如果你包含了不是由系统提供的C头文件，并且如果你能够改变的C头文件，你应该着重考虑通过添加 `extern"C"(...)` 块到头文件，这样使C++用户在C++代码中更容易使用 `#include` 。由于C编译器通不过头文件含有 `extern"C"` 的结构，你需要把 `extern"C"{}` 行包裹在 `#ifdef` 预编译块中，这样他们不会被正常的C编译器编译。

步骤#1：将以下行添加到你C头文件的顶部（注：符号 `__cplusplus` 当且仅当编译器是C++编译器的时候被定义）：

```
#ifdef __cplusplus
extern "C" {
#endif
```

步骤#2：将以下行添加到你C头文件的最底部：

```
#ifdef __cplusplus
}
#endif
```

现在您可以 `#include` 你的C头文件，不用在C++代码包含任何的 `EXTERN "C"` ：

```
// This is C++ code

// Get declaration for f(int i, char c, float x)
#include "my-C-code.h"   // Note: nothing unusual in #include line

int main()
{
  f(7, 'x', 3.14);       // Note: nothing unusual in the call
  ...
}
```

注： 对于系统提供的C头文件（如 `<cstdio>` ）和你可以更改的C头文件，准则略有不同

注： `#define` 宏有4中罪恶： 罪恶#1 , 罪恶#2 , 罪恶#3 和罪恶#4 。但有时他们仍然有用。只要别忘了使用后洗清“罪恶”的双手。

32.5 如何从C++代码中调用非系统C 函数 `f` （ `int` ， `char` 和 `float` ）？

如果你有一个个人的C函数要调用，由于一些其他原因，你没有或不想在函数声明中 `#include` 一个 C头文件，你可以在C++代码通过 `extern"C"` 语法声明单个的C函数。当然，你需要使用完整的函数原型：

```
extern "C" void f(int i, char c, float x);
```

可以使用大括号声明几个C函数：

```
extern "C" {
  void   f(int i, char c, float x);
  int    g(char* s, const char* s2);
  double sqrtOfSumOfSquares(double a, double b);
}
```

在此之后你可以象调用C++函数那样调用该函数：

```
int main()
{
  f(7, 'x', 3.14);   // Note: nothing unusual in the call
  ...
}
```

32.6 如何创建一个C++ 函数 `f` （ `int` ， `char` 和 `float` ），可以由我的C代码调用？

通过使用 `EXTERN的"C"` 结构通知C++编译器 `f（int，char,float）` 可由一个C编译器调用 ：

```
// This is C++ code

// Declare f(int,char,float) using extern "C":
extern "C" void f(int i, char c, float x);

...

// Define f(int,char,float) in some C++ module:
void f(int i, char c, float x)
{
  ...
}
```

通过 `extern"C"` 行告诉编译器应该使用C调用约定和名字校正(name mangling)（例如，以下划线开头）来进行链接。由于C不支持重载，所以你不能编写可以由C程序调用的重载函数。

32.7 为什么链接器报错说C / C++函数调用C++ / C函数？

如果你没有设置对 `EXTERN` 的 `"C"` ，你有时会得到链接错误，而不是编译器错误。这是由于C++编译器通常是“校正(mangle)”函数名称（例如，为了支持函数重载），这和C编译器不同。

关于如何使用 `EXTERN` 的 `"C"` 请参考前两个的FAQs。

32.8 如何传递一个C++ 类对象从/到一个C函数？

下面是一个例子（关于 `extern"C"` ，见前面的两个FAQs）。

```
//Fred.h:

/* This header can be read by both C and C++ compilers */
#ifndef FRED_H
#define FRED_H

#ifdef __cplusplus
  class Fred {
  public:
    Fred();
    void wilma(int);
  private:
    int a_;
  };
#else
  typedef
    struct Fred
      Fred;
#endif

#ifdef __cplusplus
extern "C" {
#endif

#if defined(__STDC__) || defined(__cplusplus)
  extern void c_function(Fred*);   /* ANSI C prototypes */
  extern Fred* cplusplus_callback_function(Fred*);
#else
  extern void c_function();        /* K&R style */
  extern Fred* cplusplus_callback_function();
#endif

#ifdef __cplusplus
}
#endif

#endif /*FRED_H*/

// Fred.cpp:

// This is C++ code

#include "Fred.h"

Fred::Fred() : a_(0) { }

void Fred::wilma(int a) { }

Fred* cplusplus_callback_function(Fred* fred)
{
  fred->wilma(123);
  return fred;
}

//main.cpp:

// This is C++ code

#include "Fred.h"
```

```
int main()
{
  Fred fred;
  c_function(&fred);
  ...
}

//c-function.c

/* This is C code */

#include "Fred.h"

void c_function(Fred* fred)
{
  cplusplus_callback_function(fred);
}
```

不像C++代码，C代码将无法告诉你两个指针是否指向同一个对象，除非指针完全相同。例如，在C++可以很容易地检查一个派生类 `Derived*` 指针 `dp` 和 `Base*` 指针 `bp` 是否指向同一个对象，你可以使用 `if(dp == bp)` 。C++编译器自动转换指针为相同的类型，在这种情况下，转换为 `Base*` ，然后比较他们。根据不同的C++编译器的实现细节，这种转换有时会改变一个指针值的位数据（bits）。但是C编译器不会知道该怎么做指针转换，所以比如从 `Derived*` 到 `Base*` 的转换，必须在由C++编译器的编译的代码中，而不是在C编译器编译的C代码中。

注意：你必须特别小心转换为 *`void*`* 指针，因为该转换将不会允许*C*或*C++*编译器做适当的指针调整！例如（继续前段内容），如果你把 *`dp`* 和 *`bp`* 赋值到两个 *`void *`* 指针比如说是 *`dpv`* 和 *`bpv`* ，有可能 *`dpv != bpv`* 即使 *`dp==bp`* 。不要说我没有提醒你！

32.9 我的C函数可以直接访问一个C++ 对象的数据吗？

有时。

（有关传递C＋＋对象到/从C函数的基本内容，阅读以前的FAQ）。

你可以安全地从C函数访问C++对象的数据，如果C++类：

- 没有虚函数（包括继承虚函数）
- 它的所有数据在相同的访问级别（私有/保护/公共）
- 虚函数没有完全包含的子对象

C++类有任何基类（或任何完全包含子对象具有基类），访问数据将在技术上是不可移植的，因为继承体系下的类的布局与语言无关。然而在实践中，所有C++编译器都使用相同的方式：首先是基类对象（多重继承按照左到右的顺序），然后是成员对象。

此外，如果类（或任何基类）含有虚函数，几乎所有C++编译器给对象添加一个 `void *` ，在第一个虚拟函数的位置或在对象的开始位置。同样，这也不是语言要求的，但几乎所有的编译器都是这样实现的。

如果类有任何虚基类，这将更复杂，更不便于移植。一个常见的实现技术是，在对象的最后位置放置一个虚基类对象（ `v` ）（无论`v`在继承层次结构中的位置）。对象的其它部分按照正常顺序布局。每一个虚基类`v`的派生类，实际上都有一个指向`v`的指针 。

32.10 为什么C++而不是为C让我觉得“更加远离机器”？

因为你就是。

作为一个面向对象编程语言，C++允许你对问题域建模，这将允许你使用问题域的语言编程，而不是解决方案域的语言。

C的优势之一是，它已“没有任何隐藏的机制”：所见即所得。你可以阅读一个 C程序，“看到”每个时钟周期。在C++中可不是这样，老的C程序员（如我们许多人曾经是），对于这个特性往往会很矛盾（也许是“敌视”？）。但当他们转变到面向对象思想以后，他们往往认识到，虽然C++的隐藏一些机制，但是它也提供了更高的抽象和更简洁的表达，从而能够在保持运行时性能的同时降低后期维护成本。

当然你可能会编写糟糕的代码不管使用任何语言，C++并不保证好的质量，可重用性，抽象，或任何“优异的”测试指标。

C++中无法阻止糟糕的程序员编写的糟糕的程序，但是它能够让优秀的开发人员创建出色的软件。

[33] 成员函数指针

FAQs in section [33]:

33.1 “成员函数指针”类型不同于“函数指针”吗？

对。

考虑下面的函数：

`int f(char a, float b);`

函数的类型不同取决于它是否是一个普通函数或某些类非静态成员函数：

- 它的类型是“int(*)(char, float）”，如果它是一个普通的函数
- 它的类型是“`int (Fred::*)(char,float)`”，如果它是类Fred的非静态成员函数

注意：如果它是类Fred的静态成员函数 ，它的类型和普通函数是相同的：“int(*)(char, float)”。

33.2 如何将一个成员函数指针传递到信号处理函数，X事件回调函数，系统调用来启动一个线程/任务等？

不要。

由于成员函数是没有意义，如果没有一个对象来触发的话，所以你不能这样直接调用（如X窗口系统代码用C++重写的话，它很可能会传递对象的引用，不仅仅是函数指针，当然了对象将包含所需的函数，甚至更多）。

作为对现有软件的补丁，使用顶级(top-level)函数（非成员函数）作为包装器，包装器接受通过一些其他技术实例化的对象为参数。取决于你要调用的函数，这个“其他技术”有可能很琐碎或者不需要你做太多的工作。对于启动一个新线程的系统调用，例如，可能要求你传递一个void 类型的函数指针，这种情况下你可以传递*void*类型的对象指针。许多实时操作系统要启动一个新任务时候于此类似。最坏的情况，你可以将对象指针存储在全局变量中，对于Unix的信号处理程序来说可能需要这样处理（但一般来说不希望使用全局变量）。在任何情况下，顶级（top-level）函数将负责调用相应的对象的成员函数。

下面是一个最坏的例子（要使用全局变量）。中断发生时候假设你要调用 `Fred::memberFn()` ：

```
class Fred {
public:
  void memberFn();
  static void staticMemberFn();  // A static member function can usually handle it
  ...
};

// Wrapper function uses a global to remember the object:
Fred* object_which_will_handle_signal;

void Fred_memberFn_wrapper()
{
  object_which_will_handle_signal->memberFn();
}

int main()
{
  /* signal(SIGINT, Fred::memberFn);& */ // Can NOT do this
  signal(SIGINT, Fred_memberFn_wrapper);  // OK
  signal(SIGINT, Fred::staticMemberFn);   // OK usually; see below
  ...
}
```

注： 静态成员函数并不需要一个实际对象来触发，因此静态成员函数指针和普通函数指针“通常”是兼容的。然而，尽管它可能在大多数编译器上工作，但是严格来说它必须是带有 `extern"C"` 修饰的非成员函数。因为“C链接器”不仅不知道“名字校正(mangle)”等，而且还不知道不同的调用约定，而C和C++的调用约定可能不同。

33.3 为什么我总是收到编译错误（类型不匹配）当我尝试用一个成员函数作为中断服务例程？

这是前两个问题的特殊情况，因此，阅读前两个FAQ问题的答案。

非静态成员函数有一个隐藏的参数，对应于 `this` 指针，该 `this` 指针指向的对象的实例。系统的中断硬件/固件不能提供有关 `this` 指针参数。你必须使用“普通”函数（非类成员）或静态成员函数作为中断服务例程。

一个可行的办法是使用一个静态成员函数作为中断服务程序，并让该静态函数去负责查找在中断时候应该调用的实例/成员函数。实际效果是，中断的时候成员函数被调用，但是出于技术原因你需要调用一个中间函数。

33.4 为什么取C++函数的地址我会遇到问题？

简单答案：如果你试图把它存储到（或者传递到）函数指针，这就会产生问题-这是前面FAQ问题的必然结果。

详细回答：在C++成员函数有一个隐含的参数，它指向对象（内部成员函数的this指针）。普通C函数和成员函数有不同的函数调用约定，所以他们的指针类型（成员函数指针与普通函数指针）是不同的，不相容的。C++中引入了新的指针类型，称为成员指针，它只能供一个实例对象调用。

注意： 不要试图强制转换成员函数指针为普通函数指针，结果是不确定的，可能是灾难性的。例如，一个成员函数指针不需要包含确切函数的机器地址。正如在最后一个例子，如果你有一个普通C函数的指针，使用一个顶层（非成员）函数或静态 （类）成员函数。

33.5 使用成员函数指针调用函数时我如何才能避免语法错误？

同时使用 `typedef` 和 `#define` 宏。

步骤**1**：创建typedef：

```
class Fred {
public:
  int f(char x, float y);
  int g(char x, float y);
  int h(char x, float y);
  int i(char x, float y);
  ...
};

// FredMemFn points to a member of Fred that takes (char,float)
typedef  int (Fred::*FredMemFn)(char x, float y);
```

第**2**步：创建一个 `#define` 宏：

```
#define CALL_MEMBER_FN(object,ptrToMember)  ((object).*(ptrToMember))
```

（通常我不喜欢 `#define` 宏，但在成员函数指针中你应该使用他们，因为他们可以提高可读性和代码的易用性。）

以下是如何使用这些功能：

```
void userCode(Fred& fred, FredMemFn memFn)
{
  int ans = CALL_MEMBER_FN(fred,memFn)('x', 3.14);
  // Would normally be: int ans = (fred.*memFn)('x', 3.14);

  ...
}
```

我强烈建议使用这些功能。在实践中，成员函数调用更比刚才复杂，可读性和代码的易写性的区别很大。comp.lang.C++不得不忍受成千上万的程序员的询问语法错误的帖子，。几乎所有这些错误都会消失如果他们使用了这些功能。

注：#define宏有4中罪恶： 罪恶#1，罪恶#2，罪恶#3 和罪恶#4 。但有时他们仍然有用。只要别忘了使用后洗清“罪恶”的双手。

33.6 如何创建和使用一个成员函数指针数组？

同时使用 `typedef` 和 `#define` 宏的前面描述，你就完成90%。

步骤**1**：创建typedef：

```
class Fred {
public:
  int f(char x, float y);
  int g(char x, float y);
  int h(char x, float y);
  int i(char x, float y);
  ...
};

// FredMemFn points to a member of Fred that takes (char,float)&
typedef  int (Fred::*FredMemFn)(char x, float y);
```

第**2**步：创建一个 `#define` 宏：

```
#define CALL_MEMBER_FN(object,ptrToMember)  ((object).*(ptrToMember))
```

现在简单地创建成员函数的指针数组：

```
FredMemFn a[] = { &Fred::f, &Fred::g, &Fred::h, &Fred::i };
```

也可以简单地调用成员函数的指针：

```
void userCode(Fred& fred, int memFnNum)
{
  // Assume  memFnNum  is between 0 and 3 inclusive:
  CALL_MEMBER_FN(fred, a[memFnNum]) ('x', 3.14);
}
```

注：#define宏有4中罪恶： 罪恶#1，罪恶#2，罪恶#3 和罪恶#4 。但有时他们仍然有用。虽然感到耻辱和负罪感，如果像宏这样的结构如果能够改进你的软件，那么就使用它。

33.7 可以转换成员函数指针为 `void *` 吗？

否！

```
class Fred {
public:
  int f(char x, float y);
  int g(char x, float y);
  int h(char x, float y);
  int i(char x, float y);
  ...
};

// FredMemFn points to a member of __Fred__ that takes (char,float)
typedef  int (Fred::*FredMemFn)(char x, float y);

#define CALL_MEMBER_FN(object,ptrToMember)  ((object).*(ptrToMember))

int callit(Fred& o, FredMemFn p, char x, float y)
{
  return CALL_MEMBER_FN(o,p)(x, y);
}

int main()
{
  FredMemFn p = &Fred::f;
  void* p2 = (void*)p;                  // ← illegal!!
  Fred o;
  callit(o, p, 'x', 3.14f);             // okay
  callit(o, FredMemFn(p2), 'x', 3.14f); // might fail!!
  ...
}
```

请不要给我发电子邮件， 如果碰巧上述情况在您的特定的操作系统和特定的编译器的特定版本中没有问题。我不在乎这些。这中做法是非法的，句号！

33.8 可以转换函数指针为 `void *` 吗？

否！

```
int f(char x, float y);
int g(char x, float y);

typedef int(*FunctPtr)(char,float);

int callit(FunctPtr p, char x, float y)
{
  return p(x, y);
}

int main()
{
  FunctPtr p = f;
  void* p2 = (void*)p;              // ← illegal!!
  callit(p, 'x', 3.14f);            // okay
  callit(FunctPtr(p2), 'x', 3.14f); // might fail!!
  ...
}
```

请不要给我发电子邮件， 如果碰巧上述情况在您的特定的操作系统和特定的编译器的特定版本中没有问题。我不在乎这些。这中做法是非法的，句号！

33.9 我需要类似函数指针的功能，但需要更多的灵活性和/或线程安全，是否有其他方法？

使用functionoid。

33.10 什么是functionoid，为什么我要使用它？

Functionoids是基于steroids的函数。严格来说比函数功能更强大，而其额外的功能解决了使用函数指针时所面临的一些（不是全部）的挑战。

让我们举一个例子说明传统函数指针的使用，然后我们将其转化为使用functionoids的例子。传统的函数指针的思想是定义一堆兼容的函数：The traditional function-pointer idea is to have a bunch of compatible functions:

```
int funct1(...params...) { ...code... }
int funct2(...params...) { ...code... }
int funct3(...params...) { ...code... }
```

然后，你通过函数指针来调用：

```
typedef int(*FunctPtr)(...params...);

void myCode(FunctPtr f)
{
  ...
  f(...args-go-here...);
  ...
}
```

有时，人们创建函数指针数组：

```
FunctPtr array[10];
array[0] = funct1;
array[1] = funct1;
array[2] = funct3;
array[3] = funct2;
...
```

在这种情况下，通过访问该数组来调用函数：

```
array[i](...args-go-here...);
```

使用functionoids，首先创建了有一个纯虚函数的的基类：

```
class Funct {
public:
  virtual int doit(int x) = 0;
  virtual ~Funct() = 0;
};

inline Funct::~Funct() { }  // defined even though it's pure virtual; it's faster thi
s way; trust me
```

然后，你可以创建三个派生类来替代3个函数：

```
class Funct1 : public Funct {
public:
  virtual int doit(int x) { ...code from funct1... }
};

class Funct2 : public Funct {
public:
  virtual int doit(int x) { ...code from funct2... }
};

class Funct3 : public Funct {
public:
  virtual int doit(int x) { ...code from funct3... }
};
```

然后，不是传递一个函数指针而是传递一个 `Funct *` 。我创建 `typedef` 称为 `FunctPtr` ,只是为了代码看起来类似以前的方法：

```
typedef Funct* FunctPtr;

void myCode(FunctPtr f)
{
  ...
  f->doit(...args-go-here...);
  ...
}
```

你可以用同样的方式来创建数组：

```
FunctPtr array[10];
array[0] = new Funct1(_...ctor-args..._);
array[1] = new Funct1(_...ctor-args..._);
array[2] = new Funct3(_...ctor-args..._);
array[3] = new Funct2(_...ctor-args..._);
...
```

首先这给出了一个functionoids比函数指针功能更强大的事实，即functionoid可以传递参数可以传递到构造函数（如上图所示的ctor - argS）而函数指针版本则没有。可以想象functionoid对象为一个freeze-dried函数调用（重点在调用这个词）。 不像一个函数指针，functionoid是（概念上）一个指向了部分被调用函数的指针。想象目前的技术，让你通过传递一部分，但是不是全部参数给一个函数，然后让你freeze-dry（部分完成）函数调用。就好像这种技术可让你使用某种神奇的指针，指针指向那个freeze-dry部分完成的函数调用。然后你通过使该指针传递其余参数，系统神奇地结合你原来传递的参数（即是freeze-dried的参数），结合函数先前计算的局部变量（被freeze-dried之前），加上所有新传递的 `args` ，从函数上次被freeze-dried的地方开始继续执行函数 。这听起来像是科幻小说，但它正是概念上functionoids可以办到的。 另外 ，它可以让你反复地使用各种不同的“剩余的参数”来“完成”freeze-dried函数调用，你要你喜欢，多少次都可以。另外 ，允许（不是必须）你改变freeze-dried的状态当调用的时候，这意味着functionoids可以记得从一个调用到下一个的信息。

好吧，让我们回到现实，我会举一两个例子来解释上面叙述的意义。

假设原有函数（在老式的函数指针样式下）采取略有不同的参数。

```
int funct1(int x, float y)
{ ...code... }

int funct2(int x, const std::string& y, int z)
{ ...code... }

int funct3(int x, const std::vector<double>& y)
{ ...code... }
```

当参数不同的时候，老式的函数指针的方法是很难凑效，因为函数调用方不知道需要传递哪些参数（呼叫者仅仅有一个函数指针，而不是函数的名称或，当参数不同的时候需要的参数个数和参数类型）（不要给我发送电子邮件，我承认你可以做到这一点，但你必须花费很多精力并且收拾残局。无论如何不要给我写邮件 –请使用functionoids代替）。

使用functionoids有时情况会好很多。由于functionoid可以看作是一个freee-dried函数调用 ，只需象上面的 `y` 和/或者 `z` 一样，可以传递它们到相应的构造函数。你还可以通过共同 `args` 参数（在上例中的 `int` 类型的 `x` 参数）到 `ctor` ，但你不必-这样做。你也可以直接传递他们到的纯虚函数 `doIt()` 。 下面假设你想传递X 到 `doIt()` 和传递 `y` 和/或 `z` 到构造函数：

```
class Funct {
public:
  virtual int doit(int x) = 0;
};
```

然后，你可以创建三个派生类，而不是三个函数：

```
class Funct1 : public Funct {
public:
  Funct1(float y) : y_(y) { }
  virtual int doit(int x) { ...code from funct1... }
private:
  float y_;
};

class Funct2 : public Funct {
public:
  Funct2(const std::string& y, int z) : y_(y), z_(z) { }
  virtual int doit(int x) { _...code from funct2..._ }
private:
  std::string y_;
  int z_;
};

class Funct3 : public Funct {
public:
  Funct3(const std::vector<double>& y) : y_(y) { }
  virtual int doit(int x) { _...code from funct3..._ }
private:
  std::vector<double> y_;
};
```

当你创建的functionoids数组的时候，构造函数的参数被freeze-dried到functionoid：

```
FunctPtr array[10];

array[0] = new Funct1(3.14f);

array[1] = new Funct1(2.18f);

std::vector<double> bottlesOfBeerOnTheWall;
bottlesOfBeerOnTheWall.push_back(100);
bottlesOfBeerOnTheWall.push_back(99);
...
bottlesOfBeerOnTheWall.push_back(1);
array[2] = new Funct3(bottlesOfBeerOnTheWall);

array[3] = new Funct2("my string", 42);

...
```

因此，当用户在调用这些functionoids的 `doIt()` 的时候，他提供的“剩余” `args` ，函数调用会把传递到构造函数与传递到 `doIt()` 的参数结合起来：

```
array[i]->doit(12);
```

正如我以前说的，functionoids的优点之一是，你可以有多个实例，比方说在你的数组里面Funct1，这些实例可以有不同的参数，被freeze-dried到构造函数。例如， 数组 `[0]` 和数组 `[1]` 的类型都是 `Funct1` ，但数组 `[0] -> doIt（12）` 的行为和数组 `[1] ->doIt（12）` 的行为是不一样的，因为这将取决于传递给调用 `doIt( )`函数的12 和传递给构造函数的 `args` 。

如果我们把functionoids数组的例子变为一个本地的functionoid，你将会看到functionoids的另一个优点。为了热身，让我们回到老式的函数指针的方法，想象你要传递一个比较函数到 `sort()` 或 `binarySearch()` 例程。 `sort()` 或 `binarySearch()` 例程被称作 `childRoutine()` 和比较函数指针类型被称为 `FunctPtr` ：

```
void childRoutine(FunctPtr f)
{
  ...
  f(...args...);
  ...
}
```

然后，不同的调用方根据自己的判断传递不同的函数指针：

```
void myCaller()
{
  ...
  childRoutine(funct1);
  ...
}

void yourCaller()
{
  ...
  childRoutine(funct3);
  ...
}
```

我们可以很容易地转化为一个使用functionoids的例子：

```
void childRoutine(Funct& f)
{
  ...
  f.doit(_...args..._);
  ...
}

void myCaller()
{
  ...
  Funct1 funct(_...ctor-args..._);
  childRoutine(funct);
  ...
}

void yourCaller()
{
  ...
  Funct3 funct(_...ctor-args..._);
  childRoutine(funct);
  ...
}
```

鉴于这样的例子，我们可以看到functionoids优于函数指针的两个好处。上面讲述了在“ctor args”的好处，再加上functionoids能够在一个线程安全的环境下保持调用之间的状态。与普通的函数指针相比，人们通常通过使用静态数据来保持状态，不过静态数据是在本质上不是线程安全的---所有线程共享静态数据。但是functionoid方法本质上是线程安全的，因为这些代码是与线程本地数据想关联的。实现是很琐碎的：改变老式的静态数据为一个functionoid对象实例；并且该实现可以证明数据不仅是线程局部的，而且也可以安全的进行递归调用：每次调用 `yourCaller()` 将有自己独特的有自己独特的数据成员的 `Funct3` 对象实例。

请注意，我们已经得到了一些东西，但是不用付出任何代价。如果你想线程全局的数据，functionoids可以实现：只需更改的实例数据成员为functionoid的静态成员，或者局部范围的静态数据。该实现和函数指针相比是伯仲之间。

functionoid为你提供了第三种选择，而老式的函数指针方法却不行：允许functionoid的调用方决定他们是否希望线程局部或线程全局的数据。如果调用方希望线程全局的数据，他们需要负责的线程安全，至少他们可以有这个选择。这很容易：

```
void callerWithThreadLocalData()
{
  ...
  Funct1 funct(...ctor-args...);
  childRoutine(funct);
  ...
}

void callerWithThreadGlobalData()
{
  ...
  static Funct1 funct(...ctor-args...);  ← the static is the only difference
  childRoutine(funct);
  ...
}
```

Functionoids不能解决遇到的每一个问题当需要编写柔性软件的时候，但严格来讲他们比函数指针功能更强大，至少需要评估一下。事实上，你可以很容易证明functionoids拥有函数指针的所有功能，因为可以想像，老式函数指针相当于一个全局的（！）functionoid对象。既然你总是可以定义functionoid全局对象，你自然没有失去任何东西。证毕！

33.11 可以让functionoids快于正常的函数调用吗？

是。

如果你有一个非常小的functionoid，并在实际应用中的相当常见，函数调用本身的成本可能会很高，与由functionoid完成工作的成本相比。在以前的FAQ中，functionoids的实现使用了虚函数，这通常会花费一个函数调用成本。另一种方法使用的模板 。

下面的例子与以前的FAQ类似。我把调用 `doIt()` 修改为运算符 `()()` 来改善代码的可读性，也允许别人传递普通函数指针：

```
class Funct1 {
public:
  Funct1(float y) : y_(y) { }
  int operator()(int x) { ...code from funct1... }
private:
  float y_;
};

class Funct2 {
public:
  Funct2(const std::string& y, int z) : y_(y), z_(z) { }
  int operator()(int x) { ...code from funct2... }
private:
  std::string y_;
  int z_;
};

class Funct3 {
public:
  Funct3(const std::vector<double>& y) : y_(y) { }
  int operator()(int x) { ...code from funct3... }
private:
  std::vector<double> y_;
};
```

这种做法，在以前的FAQ的区别是fuctionoid在编译时而不是在运行时被“绑定”。想象你把它作为一个参数传递：如果你在编译时已经知道你最终要传递的functionoid，那么你可以使用以上技术，至少在典型的情况下](inline-functions.html#faq-9.3)你可以获得一个相对速度优势，就是编译器[内联代码到调用方。下面是一个例子：

```
template <typename FunctObj>
void myCode(FunctObj f)
{
  ...
  f(...args-go-here...);
  ...
}
```

编译器编译上面代码的时候，有可能内联展开的函数调用，即可能提高性能。

下面是一种调用方法：

```
void blah()
{
  ...
  Funct2 x("functionoids are powerful", 42);
  myCode(x);
  ...
}
```

补充：正如在上文第一段所述，你也可以传递普通函数（尽管调用方调用时可能会招致一些花销）：

```
void myNormalFunction(int x);

void blah()
{
  ...
  myCode(myNormalFunction);
  ...
}
```

33.12 functionoid和仿函数(functor)有什么区别？

functionoid是一个对象，有一个主要方法。它基本上是C函数的面向对象扩展，人们会使用functionoid当函数有多个入口点（即不止一个“method”），和/或者需要以线程安全的方式（C风格的解决办法是，增加一个本地的“静态”变量，但在多线程环境中不能保证线程安全）调用之间保持状态。

functor是functionoid的特殊情况：这是一个其方法是“函数调用操作符”(`operator()()`)的functionoid. 由于它重载函数调用操作符，代码可以使用和函数调用相同的语法来调用它的主体方法。例如，如果“`foo`”是一个functor，要调用“`foo`”对象的“`operator()()`”可以使用“`foo()`”。在这样的好处在于模板，模板可以有一个可以作为函数使用的模板参数，这个参数可以是一个函数或仿函数对象。它有一个性能优势，就是仿函数对象的 “`operator()()`"方法可以被内联（如果你传递一个函数地址，那么它不能被内联）。

这是非常有用的，比如对于排序容器“比较”函数。在C中，比较函数总是通过指针传递（例如，参见 “`qsort()`"声明），但在C++中参数可以是函数指针或者functor对象，其导致的结果就是C++的排序容器在某些情况下，要比C语言中的更快（不慢）。

由于Java没有任何类似模板的功能，它必须使用动态绑定，动态绑定必然意味着函数调用。这通常不是什么大问题，但在C++中，我们要让代码发挥最高性能，也就是说，C++中有一个“pay for it only if you use it”的理念，这意味着语言绝对不能随意施加任何开销到物理机器（当然是程序员有可能会，比如选择的使用如动态绑定等技术，施加一些开销，这是作为的灵活性或其他“特性”的交换，应该由设计师和程序员来决定他们是否想要这些结构带来的好处（和成本等）。

[35] 模板

FAQs in section [35]:

35.1 模板的设计思想是什么？

模板像是甜饼切割器，指定如何切割cookies让他们看起来大致相同（虽然Cookie由各种面团来制作，但是他们都会有相同的基本形状）。同样，类模板是描述如何建立一个类族，让所有的类看起来是基本相同；函数模板描述如何建立一个外观类似的函数族。

类模板通常用于构建类型安全的容器（although this only scratches the surface for how they can be used）。

35.2 什么是 “类模板”的语法/语义？

考虑一个容器 `类class Array`,它的行为像一个整数数组：

```
// This would go into a header file such as "__Array.h__"
class Array {
public:
  Array(int len=10)                  : len_(len), data_(new int[len]) { }
 ~Array()                            { delete[] data_; }
  int len() const                    { return len_;     }
  const int& operator[](int i) const { return data_[check(i)]; }  ← subscript operato
rs often come in pairs What's the deal with "const-overloading"?")
        int& operator[](int i)       { return data_[check(i)]; }  ← subscript operato
rs often come in pairs What's the deal with "const-overloading"?")
  Array(const Array&);
  Array& operator= (const Array&);
private:
  int  len_;
  int* data_;
  int  check(int i) const
    { if (i < 0 || i >= len_) throw BoundsViol("Array", i, len_);
      return i; }
};
```

对于浮点数数组，字符数组，`std::string` 数组，`std::string` 数组的数组等，反复重复上述步骤将很冗长乏味。

```
// This would go into a header file such as "__Array.h__"
template<typename T>
class Array {
public:
  Array(int len=10)                : len_(len), data_(new T[len]) { }
 ~Array()                          { delete[] data_; }
  int len() const                  { return len_;     }
  const T& operator[](int i) const { return data_[check(i)]; }
        T& operator[](int i)       { return data_[check(i)]; }
  Array(const Array<T>&);
  Array<T>& operator= (const Array<T>&);
private:
  int len_;
  T*  data_;
  int check(int i) const
    { if (i < 0 || i >= len_) throw BoundsViol("Array", i, len_);
      return i; }
};
```

与模板函数不同,模板类（实例化模板）在实例化时需要指明相关参数：

```
int main()
{
  Array<int>           ai;
  Array<float>         af;
  Array<char*>         ac;
  Array<std::string>   as;
  Array< Array<int> >  aai;
  ...
}
```

注意最后一个例子中的两个 `>` 之间的空格符。如果没有这个空格符，编译器会看到一个 `>>` （右移位）标记，而不是两个 `>` 。

35.3 什么是“函数模板”的语法/语义？

考虑下面函数，交换两个整型参数：

```
void swap(int& x, int& y)
{
  int tmp = x;
  x = y;
  y = tmp;
}
```

如果我们还要交换浮点数，长整形，字符串，集合，和文件系统等，我们就会疲于编写除了类型不同的相似的编码行。重复是电脑理想的工作，因此要用函数模板：

```
template<typename T>
void swap(T& x, T& y)
{
  T tmp = x;
  x = y;
  y = tmp;
}
```

对给定的类型每次我们使用swap()的时候，编译器将根据上述定义，并自动产生另外一个“模板函数”作为上述函数模板的实例化。例如：

```
int main()
{
  int         i,j;  /*...*/  swap(i,j);  // Instantiates a swap for int
  float       a,b;  /*...*/  swap(a,b);  // Instantiates a swap for float
  char        c,d;  /*...*/  swap(c,d);  // Instantiates a swap for char
  std::string s,t;  /*...*/  swap(s,t);  // Instantiates a swap for std::string
  ...
}
```

注：“模板函数”是一个“函数模板”的实例化形态。

35.4 如何确定显式调用函数模板的哪个版本？

当你调用一个函数模板时，编译器试图推断模板类型。大部分情况下，编译器可以成功的做到这一点，但有时你可能想要帮助编译器推断出正确的类型-要么是因为它不能推断出模板类型，或者是因为它会推断出错误类型。

例如，你可能会调用一个函数模板没有模板指定的参数类型，或者你可能想让编译器在选择正确的函数模板之前，迫使它对参数做一些转换（promotions）。在这些情况下，你需要明确地告诉编译器应该调用函数的模板哪个实例化。

下面是一个示例函数模板，模板参数 T没有出现在函数的参数列表中。在这种情况下， 编译器无法推断出模板参数类型在函数被调用时。

```
template<typename T>
void f()
{
  ...
}
```

若要调用该函数把 `T` 作为 `int` 或 `std::string` ，你可以这样做：

```
#include <string>

void sample()
{
  f<int>();          // type T will be int in this call
  f<std::string>();  // type T will be std::string in this call
}
```

这里是另一个函数，它的模板参数出现在函数的正式参数列表中（也就是说，编译器可以根据实际参数的类型推导出模板类型）：

```
template<typename T>
void g(T x)
{
  ...
}
```

现在如果你想强制实行参数转换，在编译器推断模板类型之前，你可以使用上述技术。例如，如果你只是简单调用 `g(42)` ，你会得到 `g<int>（42）` ，但如果你想传递42给 `g<long>()` ，你可以这样做： `g<long>（42）` 。（当然你也可以明确地转换参数，如可以 `g（long（42））` ，甚至 `g（42L）` ，当然如果这样的话本例子就没有什么意义了。）

同样，如果你调用 `g（“xyz”）` ，你最终会调用 `g<char*>（char*）` ，但如果你想调用 `std::string` 版本 `g<>()` ，你可以这样 `g<std::string>（”xyz“）` 。（同样你也可以转换参数，例如 `g(std::string(“xyz”)` ，不过那将是另一回事。）

35.5 什么是“参数化类型”？

换句话说，“类模板”。

参数化类型是一个类型，是参数化的类型或者值。 `list<int>` 是一个被另外一个类型(`int`)参数化的类型（ `List` ）。

35.6 什么是“泛型”？

还是“类模板”另一种说法。

不要 与“一般性(generality)”混淆（“一般性(generality)”这只是避免过于具体的解决方案），“泛型”是指类模板。

35.7 当模板类型 `T` 是 `int` 或 `std::string` 时，我的模板函数需要进行特殊处理。对特殊类型的T我该怎么实现模板特化？

在展示如何做到这一点之前，让我们确保你不会搬起石头砸自己的脚。对于用户来说是否该函数的行为不同？换言之，是否可以观察到的行为有实质性的不同？如果是这样，你可能是在自找苦吃，你可能迷惑用户--你最好使用不同名称的函数--不要使用模板，不要使用重载。例如，如果接受 `int` 类型的代码要插入一些东西到容器并且对结果排序，但接受 `std::string` 类型的代码要从容器中删除东西并且不对结果排序，这两个函数不应该是可以重载的函数对--他们可以观察的行为是不同的，所以他们应该有不同的函数名称。

但是，如果该函数的可观察到的行为是一致的，对于所有T类型仅仅局限在各自实现细节上的不同，那么就请继续读下去。让我们看看这方面的一个例子（仅仅是概念上，不是C++代码）：

```
template<typename T>
void foo(const T& x)
{
  switch (typeof(T)) {  ← conceptual only; not C++
    case int:
      ...  ← implementation details when T is int
      break;

    case std::string:
      ...  ← implementation details when T is std::string
      break;

    default:
      ...  ← implementation details when T is neither int nor std::string
      break;
  }
}
```

解决上述问题的办法就是是通过模板特化。不要使用 `switch` 语句，你需要把代码分解成单独的函数。第一个函数是默认的情况--当 `T` 是 `int` 或 `std::string` 以外的任何其他类型时候的代码：

```
template<typename T>
void foo(const T& x)
{
  ...  ← implementation details when T is neither int nor std::string
}
```

下一步是两个特例，第一个是 `int` 特例 的代码：

```
template<>
void foo<int>(const int& x)
{
  ...  ← implementation details when T is int
}
```

接着是`std::string`特例 的代码：

```
template<>
void foo<std::string>(const std::string& x)
{
  ...  ← implementation details when T is std::string
}
```

好啦，大功告成！编译器将自动选择正确的特例实现根据所使用的`T`的类型。

35.8 哈？你能提供一个具体的模板特化的例子吗？

可以。

下面我个人使用模板特化的几种常见情况是字符串化。我通常使用模板， 将不同类型的对象字符串化，但通常需要字符串化某些特定的类型，例如当字符串化 布尔变量的时候，我喜欢用“true”与“false”来代替“1”和“0”，所以当 T 是布尔类型时，我使用std::boolalpha 。此外，我喜欢浮点输出包含所有的数字（这样我就可以看得很小的差异，等等），因此当 T是一个浮点类型时候，我使用`std::setprecision`。最终的结果通常如下所示：

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <limits>

template<typename T> inline std::string stringify(const T& x)
{
  std::ostringstream out;
  out << x;
  return out.str();
}

template<> inline std::string stringify<bool>(const bool& x)
{
  std::ostringstream out;
  out << std::boolalpha << x;
  return out.str();
}

template<> inline std::string stringify<double>(const double& x)
{
  const int sigdigits = std::numeric_limits<double>::digits10;
  // or perhaps std::numeric_limits<double>::max_digits10 if that is available on you
r compiler
  std::ostringstream out;
  out << std::setprecision(sigdigits) << x;
  return out.str();
}

template<> inline std::string stringify<float>(const float& x)
{
  const int sigdigits = std::numeric_limits<float>::digits10;
  // or perhaps std::numeric_limits<float>::max_digits10 if that is available on your
compiler
  std::ostringstream out;
  out << std::setprecision(sigdigits) << x;
  return out.str();
}

template<> inline std::string stringify<long double>(const long double& x)
{
  const int sigdigits = std::numeric_limits<long double>::digits10;
  // or perhaps std::numeric_limits<long_double>::max_digits10 if that is available o
n your compiler
  std::ostringstream out;
  out << std::setprecision(sigdigits) << x;
  return out.str();
}
```

从概念上来讲他们都做同样的事情：把参数字符串化。这意味着可观察的行为是一致的，因此特化不会迷惑用户。但对于 `bool` 和浮点类型，细节的实现略有不同，因此模板特化是一个好的解决方法。

35.9 但是模板函数的大部分代码是相同的，是否有办法实现模板特化并且不用重复复制所有的源代码？

是。

例如，假设你的模板函数有很多共同的代码，与类型T相关的特定代码相对很少（仅仅是概念展示;不是C++）：

```
template<typename T>
void foo(const T& x)
{
  ... common code that works for all T types ...

  switch (typeof(T)) {  ← conceptual only; not C++
    case int:
      ... small amount of code used only when T is int ...
      break;

    case std::string:
      ... small amount of code used only when T is std::string...
      break;

    default:
      ... small amount of code used when T is neither int nor std::string ...
      break;
  }

  ... more common code that works for all T types ...
}
```

如果盲目地跟从模板特化FAQ的建议，你最终将需要重复`switch`语句之前和之后的所有代码。两全其美的方式—既不重复相同代码又可以实现`T`的特定代码，是分离`switch`语句到一个单独的函数`foo_part()`，并使用模板特殊化：

```
template<typename T> inline void foo_part(const T& x)
{
  ... small amount of code used when T is neither int nor std::string ...
}

template<> inline void foo_part<int>(const int& x)
{
  ... small amount of code used only when T is int ...
}

template<> inline void foo_part<std::string>(const std::string& x)
{
  ... small amount of code used only when T is std::string ...
}
```

主要的`foo()`函数是一个简单的模板-没有特化。请注意，`switch`语句已经被替换为`foo_part()`调用：

```
template<typename T>
void foo(const T& x)
{
  ... common code that works for all T types ...

  foo_part(x);

  ... more common code that works for all T types ...
}
```

正如你所看到的，`foo()`的函数体本身并没有任何特殊，这一切都会自动的被调用。编译器自动生成的基于`T`类型 的`foo()`，并会生成正确的`foo_part`函数，根据实际编译时的`x`的参数类型。合适的`foo_part`的特化会被实例化。

35.10 所有这些模板和模板特化都会降低程序执行速度，对不对？

错误的。

这与实现代码的质量有关，结果可能会有所不同。但是不会有任何降低。模板可能会些微影响编译速度，但一旦类型在编译时被确定，它通常会生成和非模板函数（包括内联展开等）一样快的代码。

35.11 因此模板重载了函数，对不对？

是也不是。

函数模板参与重载函数的名称解析，但规则是不同的。对于模板重载，类型需要完全匹配。如果类型不完全匹配，类型不会被转换，函数模板从可行的函数集合中被排除。这就是所谓的“SFINAE”- Subsitution Failure Is Not An Error。例如：

```
#include <iostream>
#include <typeinfo>

template<typename T> void foo(T* x)
{ std::cout << "foo<" << typeid(T).name() << ">(T*)\n"; }

void foo(int x)
{ std::cout << "foo(int)\n"; }

void foo(double x)
{ std::cout << "foo(double)\n"; }

int main()
{
    foo(42);        // matches foo(int) exactly
    foo(42.0);      // matches foo(double) exactly
    foo("abcdef");  // matches foo<T>(T*) with T = char
    return 0;
}
```

在这个例子中， 在main()函数中第一或第二次调用 `foo` 不是对 `foo<T>` 的调用，因为无论42还是42.0都没有提供给编译器的任何信息来推断 。然而第三个调用，包括 `foo<T>` 并且 `T = char` ，因此它会调用 `foo<T>` 。

35.12 为什么不能分开模板的声明和定义，把定义放到 `.cpp` 文件中？

如果你想知道的是只是如何解决这种情况，请阅读下面得两个s。但是，为了理解要那样，首先接受这些事实：

1. 模板是不是一个类或函数。 模板是一个“模式”，编译器用来生成的相似的类或者函数。

2. 为了让编译器生成的代码，它必须同时看到模板的定义（不只是声明）和特定类型/任何用于“fill in”模板的类型。例如，如果你想使用一个 `foo<int>` ，编译器必须同时看到foo模板和你要调用具体的 `foo<int>` 。
3. 编译器可能不记得另外一个 `.cpp` 文件的细节，当编译其他 `.cpp` 文件的时候。它可以，但大多数都没有，如果你正在阅读本FAQ，它几乎肯定不会。顺便说一句，这就是所谓的“独立编译模型”。

现在，基于这些事实，下面是一个范例，它表明为什么是这个样子。假设你有一个这样的模板 `Foo` 声明：

```
template<typename T>
class Foo {
public:
  Foo();
  void someMethod(T x);
private:
  T x;
};
```

类似地，模板成员函数的定义：

```
template<typename T>
Foo<T>::Foo()
{
  ...
}

template<typename T>
void Foo<T>::someMethod(T x)
{
  ...
}
```

现在，假设在文件 `Bar.cpp` 的一些代码要使用 `foo<int>` ：

```
// Bar.cpp

void blah_blah_blah()
{
  ...
  Foo<int> f;
  f.someMethod(5);
  ...
}
```

显然，某人某地将不得不调用“模式”的构造函数，和 `someMethod()` 函数以及做 `T` 为 `int` 的实例化。但是，如果你把构造函数和 `someMethod()` 的定义放到文件 `Foo.cpp` ，当编译 `Foo.cpp` 时，编译器将看到模板代码；当编译 `Bar.cpp` 时，编译器将看到 `foo<int>` 。但任何时候决不会同时看到模板代码和 `foo<int>` 。因此，通过上面的2号规则，它根本不会产生 `foo <int>::someMethod()` 的代码。

写给专家们的话：很明显我对以上内容作了简化。这是有意为之，所以请不要大声抱怨。如果你知道 `.cpp` 文件和编译单元的差别，类模板和模板类的差别，模板其实不只是美化的宏等，请不要抱怨：这个问题/解答不是为你而设。我简化它是为了新手能够“理解它”，即使这样可能会冒犯一些专家。

提醒：欲知解决方案，请阅读下面得两个 FAQs。

35.13 如何避免模板函数的链接错误？

当编译模板函数的 `.cpp` 文件的时候告诉C++编译器应该使用哪个实例。

例如，考虑 `foo.h` 头文件包含以下模板函数声明：

```
// File "foo.h"
template<typename T>
extern void foo();
```

现在假设文件 `foo.cpp` 实际上定义的模板函数：

```
// File "foo.cpp"
#include <iostream>
#include "foo.h"

template<typename T>
void foo()
{
  std::cout << "Here I am!\n";
}
```

假设文件 `main.cpp` 中使用这个模板函数通过调用 `foo<int>()` ：

```
// File "main.cpp"
#include "foo.h"

int main()
{
  foo<int>();
  ...
}
```

如果你编译和（试图）链接这两个 `.cpp` 文件，大多数编译器将生成链接错误。有三种的解决方案。第一个解决方案是物理上在 `.h` 文件中定义，即使它不是一个内联函数。这种解决办法可能（或可能不会！）造成重大代码膨胀，意味着可执行文件的大小可能会显显著增加（或者，如果你的编译器足够聪明，可能不会这么做）。

另一个解决办法是保留定义在 `.cpp` 文件中，只添加行 `template void foo<int>()` 到 `.cpp` 文件：

```
// File "foo.cpp"
#include <iostream>
#include "foo.h"

template<typename T> void foo()
{
  std::cout << "Here I am!\n";
}

template void foo<int>();
```

如果你不能修改`foo.cpp`，只需创建一个新的`.cpp`文件，例如`foo-impl.cpp`如下：

```
// File "foo-impl.cpp"
#include "foo.cpp"

template void foo<int>();
```

请注意，`foo-impl.cpp`文件包含`.cpp`文件，而不是`.h`文件。如果你觉着这样很乱，跳个踢踏舞，想想堪萨斯，跟着我重复，“我要这么做即使它很混乱。”你需要信任我。如果不信任或者致使好奇，前面的FAQ给出了理由。

35.14 如何使用C++的关键字`export`来避免模板链接错误？

C++关键字`export`是设计用来消除包含一个模板定义（无论是在头文件中或通过实现文件中）的需要。但是，在写这篇文章时，支持此功能的唯一的知名编译器，是Comeau C++。`export`关键字未来还是个未知数。说句公道话，一些编译器厂商表示他们可能永远不会实现它，而C++标准委员会已决定大家自己定夺。

在不支持关键字`export`的编译器上，如果你希望你的代码可以通过编译，并且还希望能够有效利用支持`export`关键字的编译器。你可以这样定义模板头文件：

```
// File Foo.h

template<typename T>
class Foo {
  ...
};

#ifndef USE_EXPORT_KEYWORD
  #include "Foo.cpp"
#endif
```

并定义非内联函数的源代码文件如下：

```
// File Foo.cpp

#ifndef USE_EXPORT_KEYWORD
  #define export /*nothing*/
#endif

export template<typename T> ...
```

然后，如果/当你的编译器支持 `export` 关键字的时候，并且因为某些原因你想利用该功能，只要定义符号 `USE_EXPORT_KEYWORD` 即可。

要诀就是，你现在可以开发程序， 好像你的编译器已经实现了 `export` 关键字。如果/当你的编译器眞正支持该关键字的时候，只需要定义 `USE_EXPORT_KEYWORD` 标志，重新编译，马上你就可以利用该功能。

35.15 如何避免模板类的链接错误？

当编译模板类的 `.cpp` 文件得手告诉你的C++编译器应该使用哪个模板实例。（如果你已经阅读以前的问题，答案是完全一样的，所以你也许可以跳过此答案。）

作为一个例子，考虑 `Foo.h` 头文件包含以下模板类。请注意， `Foo<T>::f()` 方法是内联的，而 `Foo<T>::g()` 和 `Foo<T>::h()` 却不是。

```
// File "Foo.h"
template<typename T>
class Foo {
public:
  void f();
  void g();
  void h();
};

template<typename T>
inline
void Foo<T>::f()
{
  ...
}
```

现在，假设文件 `Foo.cpp` 实际定义了非内联的 `Foo<T>::g()` 和 `Foo<T>::h()` ：

```
// File "Foo.cpp"
#include <iostream>
#include "Foo.h"

template<typename T>
void Foo<T>::g()
{
  std::cout << "Foo<T>::g()\n";
}

template<typename T>
void Foo<T>::h()
{
  std::cout << "Foo<T>::h()\n";
}
```

假设文件 `main.cpp` 使用该模板创建一个 `Foo<int>` 并调用其方法：

```
// File "main.cpp"
#include "Foo.h"

int main()
{
  Foo<int> x;
  x.f();
  x.g();
  x.h();
  ...
}
```

如果你编译和（试图）链接这两个 `.cpp` 文件，大多数编译器将生成链接错误。有三种的解决方案。第一个解决方案是物理上在 `.h` 文件中定义，即使它不是一个内联函数。这种解决办法可能（或可能不会！）造成重大代码膨胀，意味着可执行文件的大小可能会显显著增加（或者，如果你的编译器足够聪明，可能不会这么做）。

另一个解决办法是保留定义在 `.cpp` 文件中，只添加行 `template class Foo<int>;` 到 `.cpp` 文件：

```
// File "Foo.cpp"
#include <iostream>
#include "Foo.h"

...definition of Foo<T>::f() is unchanged -- see above...
...definition of Foo<T>::g() is unchanged -- see above...

template class Foo<int>;
```

如果你不能修改 `foo.cpp` ，只需创建一个新的 `.cpp` 文件，例如 `foo-impl.cpp` 如下：

```
// File "Foo-impl.cpp"
#include "Foo.cpp"

template class Foo<int>;
```

请注意， `foo-impl.cpp` 文件包含 `.cpp` 文件，而不是 `.h` 文件。如果你觉着这样很乱，跳个踢踏舞，想想堪萨斯，跟着我重复，“我要这么做即使它很混乱。” 你需要信任我。如果不信任或者致使好奇，前面的FAQ给出了理由。

如果你使用Comeau C++，你可能使用 `export` 关键字实现类似功能。

35.16 为什么我收到链接错误 ，当我使用模板友元的时候？

由于模板友类的复杂性。下面是一个常见的例子：

```
#include <iostream>

template<typename T>
class Foo {
public:
  Foo(const T& value = T());
  friend Foo<T> operator+ (const Foo<T>& lhs, const Foo<T>& rhs);
  friend std::ostream& operator<< (std::ostream& o, const Foo<T>& x);
private:
  T value_;
};
```

当然在某个地方我们会用到模板：

```
int main()
{
  Foo<int> lhs(1);
  Foo<int> rhs(2);
  Foo<int> result = lhs + rhs;
  std::cout << result;
  ...
}
```

当然，在某个地方需要定义各成员和友元函数：

```
template<typename T>
Foo<T>::Foo(const T& value = T())
  : value_(value)
{ }

template<typename T>
Foo<T> operator+ (const Foo<T>& lhs, const Foo<T>& rhs)
{ return Foo<T>(lhs.value_ + rhs.value_); }

template<typename T>
std::ostream& operator<< (std::ostream& o, const Foo<T>& x)
{ return o << x.value_; }
```

一个潜在问题是编译器如何理解类声明中的friends行。在看到friends行的时候，它还不知道友元函数本身也是模板，它假定他们不是模板函数，就像下面这样：

```
Foo<int> operator+ (const Foo<int>& lhs, const Foo<int>& rhs)
{ ... }

std::ostream& operator<< (std::ostream& o, const Foo<int>& x)
{ ... }
```

当你调用运算符 `+` 或运算符 `<<` 的时候，这种假设导致编译器生成一个对非模板函数的调用，但是链接器会给你一个“未定义的外部函数”错误，因为你从来没有真正的定义这些非模板函数。

解决的办法是在编译器编译类体的时候，让编译器知道运算符 `+` 和运算符 `<<` 本身是模板。有几种方法可以做到这一点；一个简单的方法是在定义函数模板类 Foo的时候预先声明模板友元：

```
template<typename T> class Foo;  // pre-declare the template class itself
template<typename T> Foo<T> operator+ (const Foo<T>& lhs, const Foo<T>& rhs);
template<typename T> std::ostream& operator<< (std::ostream& o, const Foo<T>& x);
```

在 `frend` 行中你也需要加入 `<>` ，如下所示：

```
#include <iostream>

template<typename T>
class Foo {
public:
  Foo(const T& value = T());
  friend Foo<T> operator+ <> (const Foo<T>& lhs, const Foo<T>& rhs);
  friend std::ostream& operator<< <> (std::ostream& o, const Foo<T>& x);
private:
  T value_;
};
```

这些写法将有助于编译器更好地了解友元函数。值得一提的是，它会发现友元函数本身是模板。这消除了混乱。

另一种方法是在类中同时声明和定义该友元函数。例如：

```
#include <iostream>

template<typename T>
class Foo {
public:
  Foo(const T& value = T());

  friend Foo<T> operator+ (const Foo<T>& lhs, const Foo<T>& rhs)
  {
       ...
  }

  friend std::ostream& operator<< (std::ostream& o, const Foo<T>& x)
  {
       ...
  }

private:
  T value_;
};
```

35.17 怎么理解这些繁杂的模板错误信息？

这里有一个免费工具，可以转换错误信息便于理解。在撰写本文的时候，它工作用于下列编译器：Comeau C +，Intel C++，CodeWarrior C++，gcc，Borland C++，Microsoft Visual C++和EDG C++。

这里有一个例子，下面是一些原始的gcc的错误信息：

```
rtmap.cpp: In function int main()':
rtmap.cpp:19: invalid conversion from int' to
   std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:19:   initializing argument 1 of std::_Rb_tree_iterator<_Val, _Ref,
   _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
   std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
   std::pair<const int, double>*]'
rtmap.cpp:20: invalid conversion from int' to
   std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:20:   initializing argument 1 of std::_Rb_tree_iterator<_Val, _Ref,
   _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
   std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
   std::pair<const int, double>*]'
E:/GCC3/include/c++/3.2/bits/stl_tree.h: In member function void
   std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::insert_unique(_II,
    _II) [with _InputIterator = int, _Key = int, _Val = std::pair<const int,
   double>, _KeyOfValue = std::_Select1st<std::pair<const int, double> >,
   _Compare = std::less<int>, _Alloc = std::allocator<std::pair<const int,
   double> >]':
E:/GCC3/include/c++/3.2/bits/stl_map.h:272:   instantiated from void std::map<_
Key, _Tp, _Compare, _Alloc>::insert(_InputIterator, _InputIterator) [with _Input
Iterator = int, _Key = int, _Tp = double, _Compare = std::less<int>, _Alloc = st
d::allocator<std::pair<const int, double> >]'
rtmap.cpp:21:   instantiated from here
E:/GCC3/include/c++/3.2/bits/stl_tree.h:1161: invalid type argument of unary *
   '
```

以下是经过过滤的错误信息（注：你可以配置工具让它显示更多的信息，下面输出的设置是剪裁信息到最少）：

```
rtmap.cpp: In function int main()':
rtmap.cpp:19: invalid conversion from int' to iter'
rtmap.cpp:19:   initializing argument 1 of iter(iter)'
rtmap.cpp:20: invalid conversion from int' to iter'
rtmap.cpp:20:   initializing argument 1 of iter(iter)'
stl_tree.h: In member function void map<int,double>::insert_unique(_II, _II)':
    [STL Decryptor: Suppressed 1 more STL standard header message]
rtmap.cpp:21:   instantiated from here
stl_tree.h:1161: invalid type argument of unary *'
```

以下是上面例子的源代码：

```
#include <map>
#include <algorithm>
#include <cmath>

const int values[] = { 1,2,3,4,5 };
const int NVALS = sizeof values / sizeof (int);

int main()
{
    using namespace std;

    typedef map<int, double> valmap;

    valmap m;

    for (int i = 0; i < NVALS; i++)
        m.insert(make_pair(values[i], pow(values[i], .5)));

    valmap::iterator it = 100;              // error
    valmap::iterator it2(100);              // error
    m.insert(1,2);                          // error

    return 0;
}
```

35.18 当模板派生类使用一个继承自模板基类的嵌套类型时，为什么出错？

你也许很吃惊，下面的代码是无效的C++代码，即使如此通过有些编译器：

```
template<typename T>
class B {
public:
  class Xyz { ... };  ← type nested in class B<T>
  typedef int Pqr;    ← type nested in class B<T>
};

template<typename T>
class D : public B<T> {
public:
  void g()
  {
    Xyz x;  ← bad (even though some compilers erroneously (temporarily?) accept it)
    Pqr y;  ← bad (even though some compilers erroneously (temporarily?) accept it)
  }
};
```

这可能会让你很伤脑筋，最好坐下来听我讲。

在函数 `D<T>::g()` 内，名字 `xyz` 和 `Pqr` 不依赖于模板参数 `T` ，所以他们被称作为nondependent名字。另一方面 `B<T>` 依赖模板参数 `T` ，因此 `B<T>` 称作*dependent*名字。

规则是这样的：当查找nondependent名字（比如 `Xyz` 和 `Pqr` ）的时候，编译器不会查找dependent基类（如 `B <T>` 中 ）。因此，编译器不知道他们甚至还存在，更不用说知道它们也是类型。

这时，程序员有时会添加前缀 `B <T>::` ，例如：

```
template<typename T>
class D : public B<T> {
public:
  void g()
  {
    B<T>::Xyz x;  ← bad (even though some compilers erroneously (temporarily?) accept
it)
    B<T>::Pqr y;  ← bad (even though some compilers erroneously (temporarily?) accept
it)
  }
};
```

可惜这也行不通，因为这些名字（你准备好了吗？坐下来？）不一定是类型。"哈?!?" ？"不是类型?!?" ？。“太搞了吧！任何傻瓜都可以看到他们是类型;只要看上一眼！”，你抗议。抱歉，事实是，他们可能不是类型。原因是，有可能是 `B<T>` 的特化，假设 `B<Foo>` ，其中 `B <Foo>::Xyz` 是一个数据成员。由于这种潜在的特化，编译器不能假设 `B<T>::Xyz` 是一个类型，直到它知道 `T` 。解决方案是通过 `typename` 关键字提示编译器：

```
template<typename T>
class D : public B<T> {
public:
  void g()
  {
    typename B<T>::Xyz x;  ← good
    typename B<T>::Pqr y;  ← good
  }
};
```

35.19 当模板派生类使用使用一个继承自模板基类的成员变量时 ，为什么出错？

你也许很吃惊，下面的代码是无效的C++代码，即使如此通过有些编译器：

```
template<typename T>
class B {
public:
  void f() { }  ← member of class B<T>
};

template<typename T>
class D : public B<T> {
public:
  void g()
  {
    f();  ← bad (even though some compilers erroneously (temporarily?) accept it)
  }
};
```

这可能会让你很伤脑筋，最好坐下来听我讲。

在函数 `D<T>::g()` 内，名字 `f` 不依赖于模板参数 `T` ，所以他们被称作为nondependent名字。另一方面 `B<T>` 依赖模板参数 `T` ，因此 `B<T>` 称作*dependent*名字。

规则是这样的：当查找nondependent名字（比如f）的时候，编译器不会查找dependent基类（如 `B <T>` 中）。

这并不意味着继承不起作用。类 `D <int>` 是仍然继承自类 `B <int>` ，编译器仍然让你可以隐式的做is- a转换（例如， `D<int>*` 到 `B <int> *` ），动态绑定仍然有效当虚函数被调用时，等等。但有一个如何查找名称的问题。

替代方案：

- 改变的 `f()` 的调用为 `this->f()` 。由于在模板中 `this` 指针一直是隐式实现的， `this->f()` 要依赖查找，因此推迟到模板实例化时，此时所有基类都会被查找。
- 在调用 `f()` 之前，插入 `using B<T>::f;` 语句。
- 改变的 `f()` 的调用为 `B <T>::f()` 。 但是请注意，如果 `f()` 是虚函数，这可能没有给你想要的东西，因为它禁止了虚函数带调用机制。

35.20 前一个问题可以暗伤我？难道编译器默认地产生错误代码？

是。

由于non-dependent类型 and non-dependent成员不会在dependent模板在基础类中搜索，编译器将搜索封闭范围，比如封闭名字空间。这可能会导致它在你没有意识到的情况下（！）做错误的事情。

例如：

```
class Xyz { ... };  ← global ("namespace scope") type
void f() { }        ← global ("namespace scope") function

template<typename T>
class B {
public:
  class Xyz { ... };  ← type nested in class B<T>
  void f() { }        ← member of class B<T>
};

template<typename T>
class D : public B<T> {
public:
  void g()
  {
    Xyz x;  ← suprise: you get the global Xyz!!
    f();    ← suprise: you get the global f!!
  }
};
```

`D<T>::g()` 内的 `Xyz` 和 `f` 将被解析为全局变量，而不是继承自类 `B <T>` ，这恐怕不是你的真正意图。

别埋怨我没有警告过你。

[36] 序列化与反序列化

FAQs in section [36]:

- [36.1]“序列化”是什么东东？
- [36.2] 如何选择最好的序列化技术？
- [36.3] 如何决定是要序列化为可读的（“文本”）还是不可读的（“二进制”）格式？ or non-human-readable ("binary") format?")
- [36.4] 如何序列化/反序列化数字，字符，字符串等简单类型？
- [36.5] 如何读/写简单类型为可读的（“文本”）格式？ format?")
- [36.6] 如何读/写简单类型为非可读的（“二进制”）格式？ format?")
- [36.7] 如何序列化没有继承层次结构的对象，并且该对象不包含指向其他对象的指针？
- [36.8] 如何序列化有继承层次结构的对象 ， 并且该对象不包含指向其他对象的指针？
- [36.9] 我如何序列化包含指向其他对象指针的对象，但这些指针是没有回路和链接的树形结构？
- [36.10] 我如何序列化包含指向其他对象指针的对象，这些指针是没有回路，但是有平凡链接的树形结构？
- [36.11] 我如何序列化包含指向其他对象指针的对象，这些指针是可能含有回路或者非平凡链接的图？
- [36.12] 序列化/ 反序列化对象的时候有什么注意事项？
- [36.13] 什么是图，树，节点，回路，链接，叶链接与内部节点链接？

36.1 “序列化”是什么东东？

它可以让你把一个对象或组对象存储在磁盘；或通过有线或无线传输到另一台计算机，然后通过反向过程：唤醒原始的对象。基本机制是把对象扁平化(flatten)为一维的字节流，然后把字节流再变成原始的对象。

.如同星际迷航中的运输机，把复杂的东西扁平化为1和0的序列，然后把1和0的序列（可能在另一个地方，另一个时间）重新构造为原有的复杂“东西”。

36.2 如何选择最好的序列化技术？

有很多很多的条件限制，实际上，涉及到技术整体连贯性的多个方面。因为我时间有限（翻译：我没有报酬），我将之简化为“使用人类可读的（“文本”）或者非人类可读的（“二进制“）格式”，由或多或少按照技术成熟度排序的5个小技巧组成。

当然，不局限于上述五技术。你可能会想最终混合的几种技术。当然你也可以随时使用比实际需求更复杂（编号更高）的技术。事实上，使用比实际需求更复杂的技术是明智的，如果你认为未来的式样变更需要更大的复杂度。所以，这份名单仅仅是一个很好的起点。

最好准备！有许多东东在这里呢！

1. 决定使用人类可读的（“文本”）还是非可读（“二进制”）格式 or non-human-readable ("binary") format?")。折中很困难。后面的FAQ会告诉 如何序列化简单类型为文本格式 format?")和 如何编写简单类型的二进制格式 format?")。
2. 使用最简单的解决方案，当序列化对象不是继承层次结构的一部分（也就是说，当他们都派生自同一个类）和不包含指向其他对象指针的时候。
3. 使用较简单的解决方案，当序列化对象是继承层次结构的一部分，但是不包含指向其他对象的指针的时候。
4. 使用第三级复杂的解决方案 ，当序列化对象包含指向其他对象指针的对象，但这些指针是没有回路和链接的树形结构的时候。
5. 使用第四级复杂的解决方案， 当序列化对象包含指向其他对象指针的对象，但这些指针是没有回路，只有叶子链接的图的时候。
6. 使用最复杂的解决方案，当 序列化对象包含指向其他对象指针的对象，这些指针是可能含有回路或者链接的图的时候。

下面是同样的信息，但是使用算法格式：

1. 第一步是要决定使用文本还是二进制格式。
2. 如果对象不是继承层次结构的一部分，不包含指针， 那么使用解决方案#1 。
3. 否则，如果对象不包含指向其他对象的指针， 那么使用解决方案#2 。
4. 否则，如果指针的图不包含回路和链接，那么使用解决方案#3。
5. 否则，如果指针的图不包含循序，并且链接是叶子链接，那么使用解决方案#4。
6. 否则，使用解决方案#5。

记住：可以随意混合/增加上述列表，如果你能够判断使用更复杂的技术可以让额外成本最小。

还有一件事：对象继承和对象含有指针在逻辑上是不相关的，因此#2比#3-5简单并没有任何理论依据。然而在实践中常常（并不总是）总是这样。所以请不要认为这些分类是金科玉律—在某种程度上他们有些武断，你可能需要混合的这些解决方案以满足你的具体情况。序列化技术领域远非几个问题和解答能解决的。

36.3 如何决定是要序列化为可读的（“文本”）还是不可读的（“二进制”）格式？

要小心。

这个问题没有“正确”回答，它取决于你的目标。下面是人类可读（“文本”） 与非人类可读的（“二进制”）格式的一些利弊：

- 文本格式是比较容易检查。这意味着你将不必编写额外的工具来调试输入和输出，你可以使用文本编辑器打开序列化输出，来检查输出内容是否正确。
- 二进制格式使用较少的CPU周期。但是，这种相关仅仅当你的应用程序是与CPU绑定，并且序列化和/或反序列化是在一个内部循环和bottleneck之中。记住：90％的CPU时间花费在10%的代码中，这意味着这将不会有任何实际意义，除非你 “CPU”使用量总是100％，你的序列化和/或反序列化代码仅仅花费100％的一小部分。
- 文本格式不用担心一些编程问题，例如sizeof、小印第安编码与大印第安编码。
- 二进制格式不用担心相邻值的分隔符，因为许多值具有固定长度。
- 文本格式可以生成更小的结果集，当大多数数值很小和你需要文本化二进制编码的时候，例如uuencode的或Base64。
- 二进制格式可以生成更小的结果集，当大多数数值很大或不需要文本化二进制编码的时候。

你也许会有补充添加其他的优缺点…重要的是要知道一鞋难合众人脚-请作出自己慎重决定。

还有：无论你如何选择，你要在每个文件/流的开头加上“magic”标签和版本号。版本号将显示格式规则。这样，如果你决定对格式做重大改变的时候，将仍然可以读取旧的软件产生的输出。

36.4 如何序列化/反序列化数字，字符，字符串等简单类型？

答案取决于使用人类可读（“文本”）还是非格式人类可读（“二进制”）格式的决定：

- 下面是如何读/写简单类型为可读的（“文本”）格式。
- 下面是如何读/写简单类型为非可读的（“二进制”）格式。

在本节几乎所有的其他FAQ中都需要上述FAQ中的讨论作为基础。

36.5 如何读/写简单类型为可读的（“文本”）格式？

阅读之前，请确保要评估人类可读和非人类可读的格式之间的平衡。这种均衡评估是非常重要的，所以不能因为上一个项目没有这么做而下意识地抵制它—一鞋难合众人脚。

如果你已明确决定使用人类可读（“文本”）格式，你应该记住这些要点：

- 你最好使用 `iostream` 的 `>>` 和 `<<` ，而不是它的 `read()` 和 `write()` 方法。 `>>` 和 `<<` 比较适合文本模式，而 `read()` 和 `write()` 则更适合二进制模式。
- 当储存数值时，你需要添加分隔符，以防止数值连在一起。一个简单的方法是在每个数

字之前添加一个空格（ `），这一数字1和数字2不会连在一起看起来像数值12。由于前导空格自动会被 >>` 操作符忽略，你不需要显示的处理空格当读入序列化的结果的时候。

- 字符串需要一些诀窍，因为你必须明确地知道什么时候该字符串结束。你不能明确地使用 `'\n'` 或 `'"'` 甚至 `'\ 0'` 来终止所有字符串，因为一些字符串可能包含这些字符。你要使用C++字符转义序列，例如，当你看到一个新行时写 `'\ '` 接着 `'n'` 等等。做了这个转换之后，你可以输出字符串一直到行尾（这意味着它们使用 `'\n'` 分割），也可以以 `'"'` 来分割字符串。
- 如果对于字符串你使用C++字符转义序列，对于16进制数一定要始终 `'\x'` 和 `'\u'` 之后使用相同的位数。我通常分别使用2为和4位。原因：如果你写入的十六进制数值较少，假如你只是使用 `stream` `<<` `"\\x"` `<<` `hex` `<<` `unsigned(theChar)`，当字符串中的下个字符碰巧是十六进制数位时就会得到错误的结果。例如，如果字符串包含 `'\F'` 之后是 `'A'`，那么你应该写入 `"\x0FA"` 而不是 `"\xFA"`。
- 对于像 `"\n"` 的字符，如果你不使用字符序转义序列，那么请确保操作系统不会搅乱你的字符串数据。特别是，如果你打开一个没有 `std::ios::binary` 的 `std::fstream` 时候，有的操作系统将会企图翻译行结束字符。
- 处理字符串数据的另一个方法是在字符串前面添加长度前缀，例如，写入“ `now is the time` ”为” `15:now is the time` ”。请注意，这可能使人很难读/写文件，因为长度值之后可能会有不可见的分隔字符。尽管这样这种方法也很有用。

请记住，这些是在本节中的其他FAQ中需要的一些基础知识。

36.6 如何读/写简单类型为非可读的（“二进制”）格式？

阅读之前，请确保要评估人类可读和非人类可读的格式之间的平衡。这种均衡评估是非常重要的，所以不能因为上一个项目没有这么做而下意识地抵制它—一鞋难合众人脚。

如果你已明确决定使用非人类可读（“文本”）格式，你应该记住这些要点：

- 请确保打开输入和输出流时使用 `std::ios::binary`。即使在Unix系统也需要这么做，因为它很容易做到并且它说明了你的意图，它也更容易移植和修改。
- 你最好使用的 `iostream` 的 `read()` 和 `write()` 方法而不是它的 `>>` 和 `<<` 运算符。`read()` 和 `write()` 更适合二进制模式，`>>` 和 `<<` 更适合文本模式。
- 如果二进制数据有可能被另外一台电脑读取，必须小心不同的计算机上的印第安编码问题（小印第安编码与大印第安编码）和 `sizeof` 问题。最简单的处理方法是，选定一种编码作为正式的“网络”格式，并创建一个包含依赖计算机实现的头文件（我通常称之为 `machine.h` ）。该头文件应该定义诸如 `readNetworkInt` 这样的内联函数（ `std::istream& istr` ）来读“网络 `int` ”类型。对于其他所有的基本类型，都要定义相应的函数。你可以以任何你想要的方式来定义这些类型的格式。例如，你可以定义一个“网络 `int` ”类型为32位的小印第安编码格式。在任何情况下， `machine.h` 内的函数将做任何必要的印第安编码转换， `sizeof` 转换等等。你要么对每台机器定义不同

的 `machine.h` ，要么在 `machine.h` 格式中使用# `ifdef` 宏。但无论如何，所有这些丑陋的编码将被放置在一个头文件，所有其余代码将会变得很清晰。注：浮点数误差的处理是最微妙最棘手的。可以做到，但你必须小心像 `NaN` ，缓冲区向上溢出和向下溢出，尾数的位数和指数等。

- 如果空间成本是个问题的话，例如你要储存序列化数据到一个小存储设备的或通过慢速链接发送序列化数据，你可以压缩流和/或你可以使用一些小技巧。最简单的是使用最少的字节存储小的数值。例如，要在只有有8位字节的流中存储无符号整数，你可以劫持每个字节的第8位，来判断是否有另一个字节。.这意味着你可以存储0 ... 127在1个字节，128 ... 16384在2个字节等等。如果平均值小于500，000，000左右，这比使用4字节存储无符号数。对于这一问题有许多其他演变方式，例如，对于排序的数值数组可以存储每个数值之差，存储极小值为unary格式等等。
- 字符串数据是棘手的，因为你必须明确地知道什么时候该字符串的本身结束。你不能明确地使用 `'\ 0'` 来结束所有字符串，回想一下 `std::string` 可以存储 `'\0'` 字符。最简单的方法是在字符串之前写入字符串长度。确保整形长度是“网络格式”，这样可以避免 `sizeo` f和印第安编码问题（参见前一帖的解决方案）。

请记住，这些是在本节中的其他FAQ中需要的一些基础知识。

36.7 如何序列化没有继承层次结构的对象，并且该对象不包含指向其他对象的指针？

这是最简单的问题，毫不奇怪它也最容易解决：

- 每个类应处理好自己的序列化和反序列化。你通常会创建一个成员函数，把对象序列化到存储体（如 `std::ostream` ），另一个成员函数来生成一个新的对象，或者修改现有对象，读取数据源（如一个 `std::IStream` ）并设置对象的成员。
- 如果你的对象本身包含另一个对象，例如一个 `Car` 对象可能有一个类型为 `Engine` 的成员变量，外层对象的 `serialize()` 成员函数应该简单地调用成员变量的相应函数来序列化。
- 使用前面描述的基本知识来以文本或者二进制格式来读/写简单类型或二进制 格式。
- 如果一个类的数据结构将来可能发生变化，在对象的序列化输出的开头类应该输出一个版本号。版本号仅仅代表序列化的格式 ，不应该递增类的版本号如果只是类的行为发生了变化。 就是说，该版本数字并不需要太花哨-通常不需要主版本号和次版本号。

36.8 如何序列化有继承层次结构的对象 ， 并且该对象不包含指向其他对象的指针？

假设你要序列化 `shape` 对象，其中shape是一个抽象类，它的派生类有 `Rectangle` , `Ellipse` , `Line` , `Text` , etc等。在shape类中你将声明一个纯虚函数 `serialize(std:: ostream&) const` ，并确保每个重写首先会输出类的身份。例

如，`Ellipse::serialize(std::ostream&) const`会输出`Ellipse`的标识符（可能只是一个简单的字符串，但也可以是下文讨论的几种方案）。

当反序列化对象时，事情有点棘手。通常首先从基类的静态成员函数如`Shape::unserialize(std::istream& istr)开始`。这是声明返回一个`Shape *`或可能是像`shape::Ptr`的智能指针。它读取类名标识符，然后使用某些创建模式来创建对象。例如，你可能有类名到对象的映射表，然后使用虚拟构造函数用法来创建对象。

这里有一个具体的例子：在基类`shape`内添加一个纯虚函数`create(std::istream&) const`，并定义只有一行的重写函数，来生成一个适当的派生类对象。例如，`Ellipse::create(std::istream& istr) const`将是`{ return new Ellipse(istr); }`。添加一个`static` `std::map<std::string,Shape*>`对象，映射类名到一个对应类的代表（又名原型）对象，例如，`Ellipse`将映射到`new Ellipse()`。函数`Shape::unserialize(std::istream& istr)`将读取类名，然后查找关联的`Shape*`并调用它的`create()`方法：`return theMap[className]->create(istr)，`如果类名不再映射表里则抛出一个异常(`if (theMap.count(className) == 0) throw ...something...`)。

映射表通常采用静态初始化。例如，如果文件`Ellipse.cpp`包含了派生类`Ellipse`的代码，它还将包含一个静态的对象，其构造函数将类添加到映射表：`theMap["Ellipse"] = new Ellipse()`。

说明和注意事项：

- 如果`Shape::unserialize()传`递类名到`create()`会增加一些弹性。特别是，这将让派生类可以使用两个或两个以上的类名，每个都有自己的“网络”格式。例如，派生类`Ellipse`可以被传递“`Ellipse`”和“`Circle`”，这有益于输出时节省空间或者另有其他原因。
- 在序列化过程中通过抛出异常来处理错误通常是最容易的。如果你想你可以返回`NULL`，但你需要把读取输入流的代码从派生类的构造函数到相应的`create()`方法中，最终结果往往是你的代码变得更复杂。
- 你必须小心`Shape::unserialize()`所使用的映射表，避免静态初始化顺序错误。这通常意味着对于映射表要使用初次使用时进行初始化的手法。
- 对于由`Shape::unserialize()`所使用的映射表，我个人更喜欢基于虚构造函数手法的命名构造函数手法—它能简化一些步骤。详细信息：我通常会在`shape`内定`typedef`，例如`typedef Shape* (*Factory)(std::istream&)`。这意味着`Shape::Factory`是一个函数指针，它接受`std::istream&`为参数并返回一个`shape*`。然后，我定义映射表为`std::map<std::string,Factory>`。最后，我使用类似`theMap"Ellipse"] = Ellipse::create`代码来生成映射表（其中`Ellipse::create(std::istream&`）是一个`Ellipse`类的静态成员函数，即命名构造函数用法）。你可能需要把`Shape::unserialize(std::istream& istr)`函数的返回值从`theMap[className]->create(istr)`变为`theMap[className](istr)。`
- 序列化一个`NULL`指针通常很容易，因为你已经输出了类标识符，你可以很容易地输出一个伪类标识符，比如`NULL`。你可能在`Shape::unserialize()`中需要一个额外`if`语句

，但是如果使用我上面的写法的话，你可以消除这种特殊情况（通常可以保持代码干净整齐）通过定义一个静态成员函数 `Shape* Shape::nullFactory(istream&) { return NULL; }` 。你也需要和其他的类一样把 `NULL` 加入到映射表： `theMap["NULL"] = Shape::nullFactory;` 。

- 如果标记类名的话，序列化形式可能会更小更快一些。例如，仅在第一次看到一个类名的时候写一个类名，在以后使用时只使用相应的整数索引。如 `std::map<std::string,unsigned> unique` 将使之变得简单：如果一个类名已经在映射表里，只用写 `unique[className]` ；否则设置一个变量 `unsigned n = unique.size()` ,写 `n` ，写类名，并设置 `unique[className] = n` 。（注：一定要复制到一个单独的变量，不要使用 `unique[className] = unique.size()` ！别怪我没警告你！）。当反序列化的时候，使用 `std::vector<std::string> unique` ，读出 `n` ，如果 `n == unique.size()` ，读出类名并将其添加到 `vector` 。无论哪种方式类名都是 `unique[n]` 。您还可以预先填充前 `N` 个最常见的类名，这样流就不需要包含任何字符串。

36.9 我如何序列化包含指向其他对象指针的对象，但这些指针是没有回路和链接的树形结构？

开始之前，你必须明白，“树”并不意味着对象存储在数据结构的树中。它只是意味着你的对象互相指向对方。没有回路是指，如果不断地从一个对象指针跟随到下一个对象，你永远不会回到一个较早的对象。你的对象不是在树的内部，它们是一棵“树”。如果你不理解这些话，在继续之前你应该阅读行话FAQ。

第二，如果图将来可能包含回路或者链接不要使用此技术。

既无回路也无链接的图非常常见，即使像组合或者修饰样的“递归组合”设计模式。例如，表示XML文档或HTML文档的对象可以表示为一个没有回路和链接的图。

序列化图的关键是忽视节点的身份 ，但注重其内容 。算法（通常递归）遍历树并且不断地输出其内容。例如，如果当前节点恰好有一个整数a，指针B，浮点数c和另一个指针d，那么你先输出整数 a， 然后递归遍历b指向的子节点，然后输出浮点数 c，最后递归遍历d指向的子节点 。（你不必以声明顺序来进行序列化/反序列化，唯一的规则是，序列化和反序列化的顺序要一致。）

反序列化时你需要一个构造函数以std::istream&为参数。上述对象的构造函数将读入一个整数并存储结果到a中，然后分配一个对象存储在指针b中（需要传递std::istream到对象的构造函数中，然它也可以读取流的内容）， 读入一个浮点数到c，最后将分配一个对象存储在指针d中 。一定要在对象中使用智能指针，否则，如果任何序列化抛出异常的话(除了第一个指针象之外)，将会出现内存泄露。

当创建对象的时候，使用命名构造函数惯用法很方便。这样做的好处是可以强制使用的智能指针。要在类 Foo中实现这一点，需要添加一个如 `FooPtr Foo::create(std::istream& istr) { return new Foo(istr); }` 的静态方法。（其

中 `FooPtr` 是一个指向 `Foo` 的只能指针）。警觉的读者会注意到这与以前FAQ讨论的技术很相像 -这两种技术是完全兼容的。

如果一个对象包含可变数目的子对象，例如， 一个 `std::vector` 类型的指针，通常的做法是在递归之前输出子对象的数目。当反序列化时，只需要读入子对象的数目，然后就可以使用一个循环来创建适当数量的子对象。

如果一个子对象有可能是 `NULL` 指针，一定要在序列化和反序列化时候做处理。这不应该是一个问题， 如果对象使用继承 ；详情见上面的解决方案。 否则，如果第一个序列化的成员有一个已知的范围，使用范围以外的东西来表示一个 `NULL` 指针。例如，如果一个序列化对象的第一个成员总是一个数字，那就使用比如 `N` 这样的非数字来表示一个 `NULL` 指针。反序列化是偶可以使用 `std::istream::peek()` 来检查 `'n` '标签。如果第一个成员没有范围，那就强加一个范围，例如，每个对象之前总是输出 `'x'` ，然后使用 `’y’` 来表示 `NULL` 。

如果一个对象内部本身包含另一个对象，而不是包含指向其他对象的指针，这与上述做法没有什么两样：你仍然需要递归子节点，就好像是通过指针一样。

36.10 我如何序列化包含指向其他对象指针的对象，这些指针是没有回路，但是有平凡链接的树形结构？

和之前一样，“树”并不意味着对象存储在数据结构的树中。它只是意味着你的对象互相指向对方。没有回路是指，如果不断地从一个对象指针跟随到下一个对象，你永远不会回到一个较早的对象。你的对象不是在树的内部，它们是一棵“树”。如果你不理解这些话，在继续之前你应该阅读行话FAQ。

如果图包含叶节点的链接，但这些链接可以很容易地通过一个简单的查找表重建，那么使用此解决方案。举例来说，像 `(3*(a+b) - 1/a)` 样的算术表达式的解析树可能会有链接，因为变量名称（如 `a` ）出现不止一次。如果你想让图使用完全相同的节点对象来表示该变量的两次出现，那么你可以可能需要这种解决方案。

虽然上述限制，不适合于那些没有任何链接的解决方案。但是它是如此的接近，因此你可以想办法套用那个解决方案。差异是：

- 序列化时候，完全忽视链接。
- 反序列化时候，创建一个查找表， 比如 `std::map<std::string,Node*>` ，映射变量名称到相关的节点。

警告：这里假设变量a的所有出现应该映射到同一个节点对象；如果情况比这复杂，即如果一部分a映射到一个对象，另一些部分映射到另一个对象，你可能需要使用一个更复杂的解决方案。

36.11 我如何序列化包含指向其他对象指针的对象，这些指针是可能含有回路或者非平凡链接的图？

警告：术语“树”并不意味着对象存储在数据结构的树中。它只是意味着你的对象互相指向对方。没有回路是指，如果不断地从一个对象指针跟随到下一个对象，你永远不会回到一个较早的对象。你的对象不是在树的内部，它们是一棵“树”。如果你不理解这些话，在继续之前你应该阅读行话FAQ。

如果图包含回路，或者包含比平凡链接解决方案中更复杂的链接，那么使用此解决方案。该解决方案处理的两个核心问题：它避免了无限循环，除了节点的内容之外还写入/读取每个节点的身份 。

一个节点的身份在不同的输出流中通常不会一致。例如，如果某个文件使用数字3代表节点 `x` ，一个不同的文件可以使用数字3代表一个不同的节点 `y` 。

有一些巧妙的方法来序列化这种图，但最容易描述的是两阶段(two-pass)算法，该算法使用一个对象的ID映射表 ，例如 `std::map<Node*,unsigned> oidMap` 。在第一阶段设置 `oidMap` ，也就是说，它构建一个对象指针到对象整数ID的映射。 通过递归图，在每个节点检查该节点是否已经在 `oidMap` ，如果没有在 `oidMap` 中，就添加节点和一个唯一的整数ID到oidMap，然后递归新的子节点。唯一的整数ID通常取 `oidMap.size()` ，如 `unsigned n = oidMap.size(); oidMap[nodePtr] = n` 。（是的，需要使用两条语句。你也必须这样做。 不要缩短为一条语句。莫怪我没有警告你！）

第二阶段遍历 `oidMap` 的所有节点，并在在每个节点输出入节点的身份（相关的整数ID）之后输入节点内容。当输出节点内容时候，如果包含指向其他节点的指针，不要遍历这些子对象，只需要输出子节点的身份（相关整数ID）。例如，当节点包含 `Node* child` ，只需输出整数 `oidMap[child]` 。第二阶段之后，该 `oidMap` 可以被丢弃。换句话说，节点 `*` 到无符号整数ID的映射通常不会活过任何给定图的序列化结束。

也有一些巧妙的方法来反序列化这种图，但在这里还是使用最容易描述的是两阶段(two-pass)算法。第一阶段设置一个含有正确类型的 `std::vector<Node*> v` 对象，但所有这些对象的子指针都是 `NULL` 指针。这意味着 `v[3]` 将指向对象ID是3的对象，但该对象内部的任何子指针将都是 `NULL` 指针。第二阶段设置对象内部的的子指针，例如， 如果 `v [3]` 有一个子指针叫 `child` ，应该指向对象ID是5的对象。在第二阶段的将 `v[3].child` 从 `NULL` 修改为 `v [5]` （显然封装可能禁止直接访问 `v[3].child` ，但最终 `v[3].child` 需要被改为 `v[5]` ）。反序列化一个给定流之后， `vector v` 通常被丢弃。换句话说，当序列化或反序列化不同的流时，对象ID（3，5等等）没有任何意义-这些数字只在一个给定流内有意义。

注意：如果你的对象包含多态性指针（*polymorphic pointers*） ，也就是基类指针可能指向派生类对象，那么使用以前描述的技术。 你还需要阅读的先前的关于处理 `NULL` 指针和版本号的一些技术。

注意：当递归遍历图的时候，你应该考虑访问者模式。因为序列化可能只是需要做递归遍历的原因之一，任何递归都需要避免无限循环。

36.12 序列化/ 反序列化对象的时候有什么注意事项？

除非在特殊情况下，不要在遍历时候改变节点数据。例如，有些人觉得通过简单地增加一个节点类的整型数据成员，他们就可以映射`Node*`到整数。有时也添加布尔型的`haveVisited`标志作为`Node`对象的另外一个数据成员。

但是 ，这将导致大量的多线程和/或性能问题。你的`serialize()`方法可能不能再为`const`，所以即使它在逻辑上只是读取节点数据，尽管它在逻辑上可以有多个线程同时读取节点，但是实际的算法却写入数据到节点。如果你理解线程和读/写冲突，你只能寄希望于你的代码更复杂更慢（你必须阻止所有的读取线程，只要任何线程对图进行操作的话）。如果你(或者后继的代码维护者)不理解线程和读/写冲突，这可能引起非常严重和非常不容易觉察的错误。读/写和写/写冲突很不容易觉察，这些错误很难测试到。坏消息！相信我！如果你不相信我，找个你信任的人。但是切莫偷工减料！

现在还有很多更多要说，例如一些特殊情况。但我已经花了太多时间。如果想了解更多信息，花一些钱吧。

36.13 什么是图，树，节点，回路，链接，叶链接与内部节点链接？

当你的对象包含指向其他对象的指针，你就有了一种计算机科学家称之为图的东东 。你的对象都存储在一个像树的数据结构里面；他们是一个像树的数据结构。

图的节点又名顶点相当于你的对象 ，图的边相当于你的对象里面的指针 。 该图是一个树的一个特例，称为带根有向图。要序列化的根对象对应于图的根节点 ，指针对应于直接边。

如果对象 x有一个指向对象 Y的指针，我们说x是Y双亲 或Y是X 的 孩子 。

图的路径是指从一个对象开始，顺着指针到另一个对象等等，可以是任意深度。如果存在一个从 x 到 z的路径，我们说，x是z的祖先 和/或z是x 的 子孙。

图的链接是指有两个或两个以上不同的路径可以到达同一对象。例如， 如果 z是x和y的孩子，则图有一个链接，z是一个链接节点 。

图的回路*(*环__*)*是指存在一个返回对象自己的路径： 如果 x有一个指向自己的指针，或指向Y的指针，Y又指向x，或者为 Y，Y指向 z，Z又指向x等。图是有环的如果有一个或多个回路，否则是无环的 。

一个内部节点是有孩子的节点。叶节点是没有孩子的节点。

正如本节中使用的那样，树是指一个带根的，有向的，无环图。请注意，树的节点也是树。

[37] 类库

FAQs in section [37]:

37.1 什么是“STL”？

STL的（“标准模板库”）是一个库，主要包括（ 非常高效）容器类，以及用于容器的迭代器和算法。

技术上来说，“STL”术语不再具有任何意义，因为它所提供的STL类以及其他标准类比如std::ostream等已被完全合并到标准库，但许多人仍然使用STL这个术语，听起来像它是一个独立的东西，所以你不妨习惯于这种说法。

37.2 哪里可以得到“STL”的拷贝？

由于一部分STL类已经是标准类库的一部分，你的编译器应该提供这些类。如果你的编译器不包括这些标准类，要么得到一个更新版本的编译器或下载下列任何一个STL类的拷贝：

- An STL site: `ftp.cs.rpi.edu/pub/stl`
- STL HP official site: `butler.hpl.hp.com/stl/`
- Mirror site in Europe: `www.maths.warwick.ac.uk/ftp/mirrors/c++/stl/`
- STL code alternate: `ftp.cs.rpi.edu/stl`
- The SGI implementation: `www.sgi.com/tech/stl/`
- STLport: `www.stlport.org`

用于GCC - 2.6.3的STL hacks是GNU libg + + 2.6.2.1或更高版本包的一部分（也可能会在较早版本）。感谢Mike Lindner。

另外，一些人使用“STL”来包括标准字符串头文件 `<string>` ，但是其他人反对这一用法。

37.3 如何才能在 `Fred*` 的STL容器比如 `std::vector<Fred*>` 中找到 `Fred` 对象？

如 `std::find_f()` 这样的STL函数可以帮助你找到容器内的 `T` 号元素。但是，如果容器存储的是指针，比如 `std::vector<Fred*>` ，这些函数将找到一个匹配 `Fred*` 的元素，但不能找到和 `Fred` 匹配的元素。

解决办法是使用一个可选的参数，指定了“匹配”功能。下面的类模板可以间接地让你比较指针所指向的对象。

```
template<typename T>
class DereferencedEqual {
public:
  DereferencedEqual(const T* p) : p_(p) { }
  bool operator() (const T* p2) const { return *p_ == *p2; }
private:
  const T* p_;
};
```

现在你可以使用此模板来找到适合的Fred对象：

```
void userCode(std::vector<Fred*> v, const Fred& match)
{
  std::find_if(v.begin(), v.end(), DereferencedEqual<Fred>(&match));
  ...
}
```

37.4 哪里可以得到如何使用STL的帮助？

这里有一些资源（排名不分先后）：

Rogue Wave's STL Guide: `www.ccd.bnl.gov/bcf/cluster/pgi/pgC++_lib/stdlibug/ug1.htm`

The STL FAQ: `butler.hpl.hp.com/stl/stl.faq`

Kenny Zalewski's STL guide: `www.cs.rpi.edu/projects/STL/htdocs/stl.html`

Mumit's STL Newbie's guide: `www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html`

SGI's STL Programmer's guide: `www.sgi.com/tech/stl/`

还有一些有益的书籍。

37.5 如何判断你是否有一个动态类型的C++类库？

- 提示#1：所有东西都派生自一个根类，通常是 `object` 。
- 提示#2：容器类（列表，堆栈，集等）是非模板。

- 提示#3：容器类（ 列表 ， 堆栈 ， 集等）插入/提取的是对象指针。你可以放入 `apple` 到容器中，但是当你提取的时候，编译器只知道它派生自 `object` ，所以你必须使用指针转换将其转换回 `apple*` ；你更好祈祷这的对象就是一个 `apple` 对象，你需要自己对自己负责。

你可以使用使用 `dynamic_cast` 来保证类型安全，但 它能做的也仅仅发生在运行时。这种编码风格是C++动态类型的精华。你调用一个函数，告诉该函数：“转换 `object` 对象为 `apple` 对象，要么或返回 `NULL` ，如果它不是 `apple` 类型”。不到运行时，你不知道会发生什么事。

当你使用模板来实现容器时，C++编译器可以静态验证90+%的应用程序的类型信息（“90 + %”是不对的，有些人声称应该是100%，需要持久化(persistence)的人得不到100%），我想要强调的是：C++泛型特性是从模板获得的，而不是继承。

37.6 什么是NIHCL？哪里可以得到它？

NIHCL是“国家卫生局类库”的简写，你可以从这里取得：`128.231.128.7/pub/NIHCL/nihcl-3.0.tar.Z`

NIHCL (有人读作"N-I-H-C-L," 其他人读作"nickel") 是Smalltalk类库的C++版本。NIHCL的动态类型可以应用在一些方面（比如：对象持久化）。但是有些地方，NIHCL的用法和C++语言的静态类型有冲突。

37.7 哪里的FTP下载到《数值分析》附属的代码？

此软件是商业软件，因此通过网络渠道获得是非法的。然而，只有30元左右。

37.8 为什么可执行文件这么大？

许多人惊讶可执行文件怎么这么大，特别是在源代码很少的情况下。例如，一个简单的“Hello World”程序肯能生成一个大于大多数人预计（40 + K字节）的可执行文件。

可执行文件的一个原因是部分的C++运行时库会被静态链接到应用程序。到底有多少被静态链接，取决于是否静态或动态链接标准库的编译器选项，也取决于你对标准库的使用量，以及实现代码会怎么分开库代码。例如， `<iostream>` 库非常大，包含有许多类和虚函数。对它的任何调用将会导致引入 `<iostream>` 的所有代码（但可能有的编译器选项，可以动态链接这些类库，在这种情况下你的程序可能较小）。

另一个可执行文件大的原因可能是如果你打开了调试功能（当然还是通过编译器选项）。如果是知名的编译器，这个选项可能会增加可执行文件的大小到10倍。

你需要查看编译器说明书或咨询供应商的技术支持来获得更详细的解答。

37.9 哪里可以得到更多的有关C++类库的信息？

你应该检查三个地方（顺序无关）：

- C++库常见问题，由 Nikki Lockecpplibs(AT)trmphrst(DOT)demon(DOT)co(DOT)uk " (NOSPAM)cpplibs(AT)trmphrst(DOT)demon(DOT)co(DOT)uk")维护，可以从框架网页和无框架网页得到。
- 你也应该检查 `www.mathtools.net/C_C__/` .他们有成堆的好东西，分门别类为60（目前）多个大类。
- 你也应该检查 `www.boost.org/` .他们有一些好东西，其中一些下一次可能得到标准化的提议。

重要：这些清单可能有遗漏。如果你正在寻找一些特定的上面没有的功能，可以试试如谷歌。同时，不要忘记提交“C++类库FAQ的提交表格”来帮助别人 。.